Operating Systems Lab

Lab # 03

Objectives:

- Understanding the fork, exec and wait syscall — Lecture#17
- Understanding the concept of process tree, fan and chain — Lecture#18
- IO Redirection — Lecture#08

Process Management

1) Explain the relationship between the parent and child processes created by *fork*(). What resources are shared between them? Write down any 2 use cases of *fork*() syscall.
2) Explain the output of the following code.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
int cpid;
cpid = fork();
if (cpid == 0)
{
printf("child here.\n");
printf("CHILD pid = %d\n", getpid());
printf("CHILD ppid = %d\n", getppid());
}
else
{
printf("parent here.\n");
printf("PARENT pid = %d\n", getpid());
printf("PARENT ppid = %d\n", getppid());
sleep(2);
}

return 0;
}
```

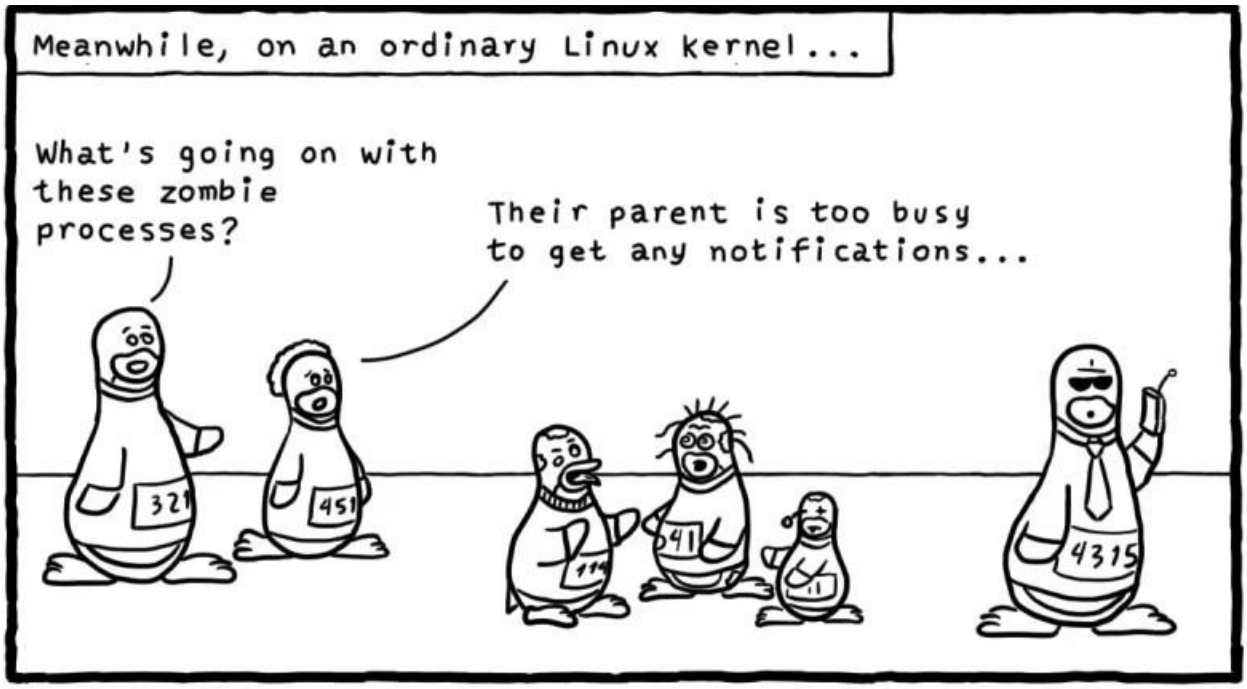
3) What will be the output of the following code.

```
#include <stdio.h>
#include<unistd.h>
#include<fcntl.h>

int main(void){

write(1, "I am learning OS", 17);
write(1, "I know what is syscall", 23);
write(1, "I am going to run the echo command", 35);
execl("/usr/bin/echo", "echo", "i am here", NULL);
write(1, "Should i be printed on screen or not", 37);
return 0;
}
```

4) Write down a C program which will print the contents of present working directories using `execlp`.



5) What are Orphan and Zombie processes? How can we see how many zombie processes there are in our system?

Check Point

6) What will be the output of this C Program:

```
int main(){
for (int i=1;i<=4;i++){
fork();
fprintf(stderr, "%s\n","ARIF");
}
exit(0);
}
```

7) What will be the output of this C Program:

```
int main(){

if (fork()||fork())

fork();

printf("1 ");

        }
```

8) What will the output of the following code? Does the parent reap the child successfully or the child becomes a zombie or orphan? Provide proof to whatever you think is the case.

```
int main(){
int cpid;
cpid = fork();
switch(cpid)
{
case 0:
printf("I am not a zombie process"); break;
default:
while(1);
}
}
```

IO Redirection

1) *What is the Per Process File Descriptor Table (PPFDT), and how does it manage file descriptors for individual processes in Unix-like operating systems?*
2) *Draw PPFDTs of **cat** and **grep** in the following command.*
   cat /etc/passwd | grep root
3) Perform the following tasks:
   a. *Write a single command to copy the contents of the file **/etc/passwd** into **output.txt** without using copy.*
   b. *Find all the files named *libc.so* in your root directory (using **find** command) and redirect the output to the file libc_locations.txt and errors to the file /dev/null.*
4) *Save the following source code as hacking.c. Compile and make executable of the hacking.c and perform I/O redirection operations as described below:*
   a) Redirect the output to a file named work_hard.txt.
   b) Redirect the error to a file named failed.txt.
   c) Redirect the stdout and stderr to a file called screen_copy.txt using copy descriptor.

```
#include <stdio.h>

#include<unistd.h>

#include<fcntl.h>

int main(void) {

int fd = open("/tmp/fake", O_RDONLY);

perror("ARM: Can't open file");

printf("Ever wanted to be a Hacker?\n");
```

```
printf("If Yes, Work hard and learn how OS throws errors to other files.\n");
return 0;
}
```

5) Interpret the meaning of following if input file f1.txt exist or does not exist
   - $ cat f1.txt > f2.txt
   - $ cat f1.txt 1>f2.txt 2> f2.txt
   - $ cat 0< f1.txt 1>> f2.txt 2>> f2.txt
   - $ cat 0< f1.txt 1>> f2.txt 2> &1
   - $ 2> errors.txt cat f1.txt > f2.txt
   - $ tee 0< f1.txt f2.txt f3.txt

Bonus Task

1) Write a C program and perform the following tasks.
   a. Your program should take exactly one command line argument
   b. Do fork and run the given cat program using execve syscall in the child process with user argument.
   c. Your parent process should wait for the child process.
   d. Once the child process is terminated, print the id of the terminated child process.