1: In the report, include only relevant lines from ps -l for your programs, and point out the following items:

- process name: CMD column, which is the last column. The processes we started are called print_pid.
- process state (decode the letter!) : the second column, called S; some of the states are R: *running*, S: *sleeping*, Z: *zombie*. Both of the processes of our program are in the state sleeping.
- process ID (PID): the fourth column. Our program was ran twice and the PID for the first one is 3371 and the second one 3372.
- parent process ID (PPID): the fifth column. Our processes both got started by the bash, so they have the PID of the bash as their PPID, which means that the PPID for our process is 3325.

```
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    3325    3144  0  80   0 -  5635 do_wai pts/4    00:00:00 bash
0 S  1000    3371    3325  0  80   0 -   622 hrtime pts/4    00:00:00 print_pid
0 S  1000    3372    3325  0  80   0 -   622 hrtime pts/4    00:00:00 print_pid
4 R  1000    3390    3325  0  80   0 -  5725 -      pts/4    00:00:00 ps
princesslighty@princesslighty-WRT-WX9:~/Skole/2.1/ITF22519/labs/lab5$ I am awake
.
I am awake.
```

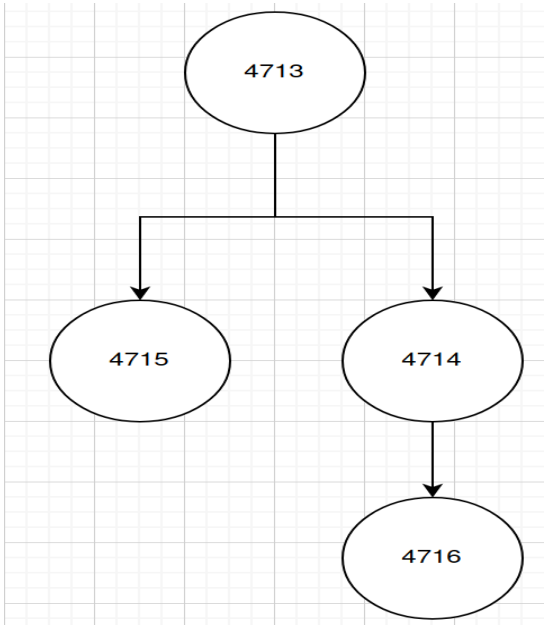
- Find out the name of the process that started your programs. What is it, and what does it do? The names are under the CMD column. The CMD column contains the command that generated the process. Looking at the table we can see that bash is the command that generated our print_pid processes. It also started the ps command under our. We can also see that ps has the same PPID as print_pid to see that they both share the same command, bash, as their parent process.

2: Compile and execute the program fork_ex1.c.

- Include the output of the program.

```
Process 4713's parent process ID is 3325
Process 4715's parent process ID is 4713
Process 4714's parent process ID is 4713
Process 4716's parent process ID is 4714
```

- Draw the process tree (label processes with PIDs).



- Explain how the tree was built. Process 4713 is the initial command that started the processes in this process tree. The first child is the process 4714 and the second one is 4715. Then 4714 becomes a parent process to 4716.

3: The program fork_ex2.c prints different messages in the child and parent processes. Complete the condition of the if statements in this program.

if (pid == 0) to find the child because if the id of the fork is 0 it's always a child.
else if (pid > 0) to find the parent because that means it is a parent since the id isn't 0.

4: Summarize the usage of wait() and waitpid().

- Wait**():** forces the parent to pause execution until one of its children terminates.
- Waitpid(): the difference between this and wait() is that this one waits for a specific child to terminate before continuing execution. The pid must be used as an argumemt of waitpid to specify which one to wait for.

```
waitpid(child, &status, 0);

wait(&status);        // is the same as waitpid(-1, &status, 0);
```

Here we see that the arguments of the commands are almost identical except for the first argument.

5: Run the program execl.c and explain the results.

Because the command execl("/bin/ls", "ls", NULL); replaces the process image with a new process image; which means that everything below the command no longer will get executed unless the command fails to execute successfully. The process ID doesn't change because it only replaces the process image, and doesn't create a new process. The first parameter, const char *path, is for the path of the file that will replace the current one, the second, const char *arg, are the arguments to be executed and the third has to be a NULL pointer. What this code actually does is that it goes to the path in the first argument to replace that file with the one currently being active in the process, so it goes into ls which is located inside the bin directory and runs the ls command which is a command that lists the contents of the current/chosen directory, in the end it returns NULL to terminate the arguments on completion.

6: The program print_args.c prints out the values stores inside the two arguments: argc and argv passed into the main function. Compile and play around with this program.

argc is an int used to store the amount of command line arguments passed by the user including the name of the program. So when you execute a program by doing ./filepath it will only be one, but if you have other arguments in the same line, it will count those other arguments in the storage. The value of argc shouldn't be negative.
argv is an array of character pointers listing all the arguments where element zero is the name of the program.
They can both be omitted as arguments in the main function if your program isn't supposed to process command line arguments, then you can just write int main().
The code in this program just prints the amount of arguments is passed, and the loop runs as many times as the value of argc; inside the loop we print all the arguments being passed.

```
princesslighty@princesslighty-WRT-WX9:~/Skole/2.1/ITF22519/labs/lab5$ ./print_args m m
argc = 3
argv[0] = ./print_args
argv[1] = m
argv[2] = m
```