1 OPERATING SYSTEMS LABORATORY IV - Processes I

1. **Examples:**

   - Compile and run the code.
   - Analyze the code and output.

   (a) Creating a (child) process; fork - code14.c.

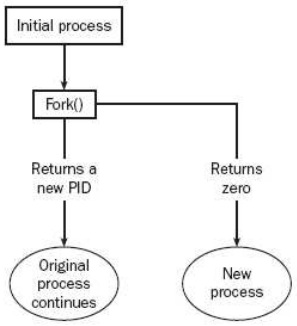

Figure 1: System call; Fork

```
#include <stdio.h>
main()
{
  puts("Begin fork test.");
  fork();
  puts("End fork test.");
}
```

You should have the message "End fork test." twice. Why?

(b) Parent and child process; getpid - code15.c.

- Execute this code several times. You should observe that the order for the print messages of the parent and child processes change. Why?

(c) Fork at the beginning and error analysis; perror - code16.c.

- Find out that the function **perror** is included by which library.
  - Why the value -1 is checked?

- Fork is executed at the beginning of the program. So that the rest of the code will be duplicated, now belong to both parent and child processes.
- Execute several times and examine the order in the output. Is there any specific order that which one will be executed first?
- What is the function of the type definition **pid_t**?

(d) Signaling; sleep,getppid - code17.c.

- What is the function of **sleep**?
- Who is the parent of the PARENT? Find it out by the command

  `ps aux | grep ParentPID`
- Why CHILD process prints out the PID of its parent as 1 (See next example)?

(e) Synchronizing; wait - code18.c.

- What is meant by synchronization?
- **wait** and **sleep** are system calls. What is the function of **wait**?
- Why CHILD process prints out the PID of its parent correctly now?!

(f) Zombie processes - code19.c.

- When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait.
- The child process entry in the process table is therefore not freed up immediately.
- Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a zombie process.
- Call the **ps -ux** program in another shell after the child has finished but before the parent has finished, we'll see a $< defunct >$ phrase in the line. (Some systems may say $< zombie >$ rather than $< defunct >$.)
- If the parent then terminates abnormally, the child process automatically gets the process with PID 1 (*init*) as parent.
- The child process is now a **zombie** that is no longer running but has been inherited by *init* because of the abnormal termination of the parent process.

(g) Understanding **system** - code20.c .

- Try to recognize your "code20" and corresponding PID from the output.

2. **Exercises:**

(a) Write a program that creates a zombie and then call **system** to execute the **ps** command to verify that the process is zombie.

(b) Write a program to create 4 processes where first process is the parent of the second one and the second process is the parent of the third one and the third process is the parent of the fourth one. Your program should be capable of;

- checking if the processes is forked with success,
- printing the pid and parent pid of each process,
- printing the parent pid of the first process.