Lab Practice Problems #1

1. Write a program that accepts an integer k as a command-line argument and spawns k child processes. The parent process should wait for each child to finish, reaping the child processes in the order they were created. After reaping each child, the parent should print the PID of the reaped child and then exit. The child should print its PID, sleep for 2 seconds, and exit.

2. Create a parent process that redirects standard output to a file using dup2(). Fork a child process. Investigate whether the file descriptor redirection is inherited by the child process as well.

3. Write a program that uses pipes to create two child processes that communicate with each other. The first child should send data to the second child, read it, and print it to stdout.

4. Write a program that takes two file names from the command line and copies the content of the first file into the second file in reverse order (last character to first character) using system calls like open, read, write, and close. Don't use IO functions provided by stdio.h like printf, fopen, fscanf, etc. You can assume that file size doesn't exceed 500 characters.

5. Create a child process from a parent process and print both parent's and child's PIDs and PPIDs. The child's PID should show the parent's PID correctly. Then, make the child process sleep for some time during which the parent process terminates and after waking up the child process prints the PID and PPID. The PPID of the child should have changed by now. Make observations on which process becomes the new parent of this child process (use ps to verify).

6. Write a program that creates three child processes; child1 will take integer as an input from stdin and write it to the pipe connected with child2, child2 takes another integer as input from stdin, and then transfers both integers to child3 using another pipe. child3 performs **addition** on the two integer values and prints result to stdout.

7. Write a C program that uses the dup2() system call to redirect the standard error stream (stderr) of a child process. The program should create a file named errors.log to write error messages. Use fork() to create a child process. In the child process, use dup2() to redirect stderr to the errors.log file. In the child process, intentionally cause an error (e.g., by trying to open a non-existent file or divide by zero) and observe how the error message is captured in errors.log. The parent process should wait for the child process to finish using wait().

8. Write a C program that does inter-process communication using pipes. The program should create two child processes using fork(). Establish a pipeline between the parent process and the first child and another between the first child and the second child using pipe(). The parent process should send ***five*** integers to the first child through the first pipe. The first child process should read the integers, calculate the square of each integer, and send the squared values to the second child through the second pipe. The second child process should read the squared integers and compute their sum. The second child should send the result (the sum of the squared integers) back to the parent process using another pipe. Finally, the parent process should read the result from the pipe and print it.

9. Write a program where an integer k (greater than 0) is provided as input. The main process of the program creates **k** child processes and prints their PID as they are created, and also a single pipe. All child processes will read from the pipe that was created by the main process. The main process writes integers from 1 to k into these pipes. The child process will try to read an integer from its pipe, print it the value and the PID of the child process, and then terminate (all simultaneously). What can be inferred from the sequence in which child processes are created versus the sequence in which they terminate? How consistent or variable is this sequence across different runs of the program? Why?

10. Write a program to demonstrate the presence of zombie processes. The program forks a process, and the processes print PID information. Subsequently, the parent process sleeps for 1 minute and then waits for the child process to exit. The child process waits for keyboard input from the user after displaying the messages and then exits. Check the process state of the child process while it is waiting for input and after it terminates. Use the ps command in a separate terminal window to check the process state.

11. Write a program that takes three filenames and an integer offset as argument, copies the content of the first file to the third file, and then additionally copies the content of the second file to the third file starting at the mentioned offset.
    The catch is that you should not use the file and IO-related functions provided via stdio.h (e.g., `scanf, printf, fopen, fread, fwrite, fprintf, fscanf, ...`).

12. `int system(const char *command)`
    is a standard library function that is used to execute a shell command.
    Write your own version of the Unix system function
    `int mysystem(char *command);`

    The mysystem function executes the `command` (passed in the function) by invoking the `/bin/sh -c command` and then returns after the `command` has been completed. If the command exits normally (by calling the exit function or executing a return statement), then mysystem returns the command exit status. For example, if the command terminates by calling exit(8), then mysystem returns the value 8. Otherwise, if the command terminates abnormally, then mysystem should return the status returned by the shell.
    The commands that can be used with the function are listed here:
    https://linuxhandbook.com/shell-builtin-commands/

13. Write a program that uses the exec system call to run the "`ls`" command. First, run the command with no arguments. Then, change your program to provide some arguments to "`ls`", e.g., "`ls -l`". In both cases, running your program should produce the same output as that produced by the "ls" command. There are many variants of the exec system call, e.g., `execvp`, `execlp`, and so on. Read through the man pages to find something that suits your needs.

14. Write a program where a process forks a child. The child runs for a long time, say, by calling the sleep function. The parent must use the kill system call to terminate the child process, reap it, print a message, and exit. Understand how signals work before you write the program.

15. Write a program that runs an infinite while loop. The program should not terminate when it receives a SIGINT (Ctrl+C) signal, but instead, it must print "I will run forever" and continue its execution. You must write a signal handler that handles SIGINT in order to achieve this. So, how do you end this program? When you are done playing with this program, you may need to terminate it using the SIGKILL signal.