COMP 304- Operating Systems: Assignment 1
Deadline: 02 March 2019, 11 pm
(50 points)

Notes: This is an individual assignment, partners are not allowed. Any form of cheating will be harshly punished! You may discuss the problems with your peers but the submitted work must be your own work. No late assignment will be accepted. Submit your answers through blackboard. This assignment is **worth 4%** of your total grade.

Corresponding TA: Najeeb Ahmad. Office Hours for Najeeb is Thursdays 16:00-17.30 in ENG 110. For homework related questions, please use the discussion forum at Blackboard.

Important Remarks:

1. All the programming assignments should be implemented and tested in a Linux Operating System.
2. **What to submit:** Create a folder with your KUSIS ID and submit the folder as a single zip file. The folder should be organized as follows:
    a. Each problem should be in a separate subfolder
    b. Each subfolder should contain .c source files, screenshots of sample runs and a single README.txt file explaining how to compile and run our code. You may include a makefile if you have one.
    c. **Do not** submit object files or executables to Blackboard.
    d. Use the problem number to name your source files. For example, p2_a.c refers to the source code solution for problem 2 part a.
    e. Improper namings or file organizations will result in **5 point penalty** in the assignment grade.

Problem 1 - Process creation (5 x 3 pts)

In Unix/Linux, new processes are created using the `fork()` system call. Calling `fork()` creates copy of the current process (in which `fork()` is called) and the new process starts executing immediately. After the `fork()` call, both parent and child process start executing the same code.

- **Part (a)** Write a C program, which calls `fork()` 4 times consecutively. Then each of the forked processes prints its ID, its parent's ID and its level in the process tree (The level of the main process is 0). Sample output for 3 forks is shown in **Figure 1.**

Base Process ID: 30511, level: 0
Process ID: 30512, Parent ID: 30511, level: 1
Process ID: 30513, Parent ID: 30511, level: 1
Process ID: 30514, Parent ID: 30511, level: 1
Process ID: 30515, Parent ID: 30512, level: 2
Process ID: 30517, Parent ID: 30512, level: 2
Process ID: 30516, Parent ID: 30513, level: 2
Process ID: 30518, Parent ID: 30515, level: 3

Figure 1

- **Part (b)** As mentioned previously, the child process is the copy of the parent process when created with **`fork()`** system call. However, it is possible to replace the contents of the child process with a new process. This is made possible through **`exec()`** family of system calls. Write a C program that forks a child process which executes **"ps f"** command. The **"ps f"** command will display a process tree. While the child process executes the command, the parent should wait for the child to finish. When the child finishes, the parent should print a message "*Child finished execution*".
- **Part (c)** Write a C program that forks a child process that immediately becomes a zombie process. This zombie process must remain in the system for at least 5 seconds. A zombie process is created when a process terminates but its parent does not invoke wait() immediately. By not invoking wait(), the child maintains its PID and an entry in the process entry table.
    - Use the sleep() system call API for the time requirement.
    - Run the program in the background (using the & parameter) and then run the command ps -l to determine whether the child is a zombie process.
    - The process states are shown below the S column; processes with a state of Z are zombies in the ps command output.
    - The process identifier (PID) of the child process is listed in the PID column, and that of the parent is listed in the PPID column.
    - You don't want to create too many zombies. If you need, you can kill the parent process with kill -9 PID.
    - Provide the source code (only .c) and the screenshots of the ps commands.

==========

Problem 2 - Ordinary pipes (15 pts)

In parallel and distributed computing, one of the important strategies is to use multiple processes cooperating with each other to solve a bigger problem. For this purpose, the processes need to exchange data among themselves. In Unix/Linux, one of the options is to use ordinary pipes for inter-process communication.

In this problem, you will use ordinary pipes to find the maximum of a list of M integers by distributing them among a set of N children processes. The parent process will create N children processes and evenly distribute the list elements among its children. Each child will then calculate the maximum in their sublist it received from the parent and send the local max to the parent. After the parent receives the local max of each child, it finds the global maximum and displays it in standard output. The program should work under the following constraints:

- The program receives M and N as command line parameters
- If M is less than N, the program prints an error message and then terminates.
- Parent process randomly generates M numbers between 0-100.
- The program should handle the case when M is not exactly divisible by the number of processes (N).

Here is a sample output:

Parent: N = 4, M = 15
Sending sublist to process 1
Sending sublist to process 2
Sending sublist to process 3
Sending sublist to process 4
Max value in Process 1 is 55
Max value in Process 2 is 21
Max value in Process 3 is 78
Max value in Process 4 is 54
Global Maximum is 78

==========

Problem 3 Shared memory communication (20 pts)

Chinese whispers (kulaktan kulağa) is a popular children's game in which the first player comes up with a message, whispers in the ear of the second player, who then whispers the message into the ear of the third player and so on. This carries on until the last player receives the message who then reads the message aloud. Most often than not, the message that is read is different than the original message by the first player.

In this problem, you will write a code that will simulate the Chinese whispers game using shared memory. You are required to implement two C programs; the first will be a shared-memory consumer-producer program while the second one will be a driver program. These two programs need to function as follows:

The driver program

- This is the program that simulates the Chinese whisper by creating N processes.
- The program accepts three command line arguments; the executable name of the "consumer-producer program", an integer N (number of processes), and a string message

- The program then forks N many children, each of which executes the **consumer-producer** program with necessary arguments to play the game.
- The **number of processes** should be at least 2. Otherwise, the program should terminate.
- Note that the driver program must create a new child only after the current child terminates to prevent concurrent writes to the shared memory.

The consumer-producer program

- If this program is run by the first child of the driver program, then it creates a shared memory segment and writes the original message into it.
- Otherwise, the program reads the message in the shared segment, randomly chooses two characters in the message, swaps those two characters and then writes back the distorted message in the shared segment. If it is run by the last child, then it also unlinks the segment.

  Here is a sample output:
  Parent: Playing Chinese whisper with 4 processes.
  Child 1: The OS assignment deadline is approaching.
  Child 2: Tae OS hssignment deadline is approaching.
  Child 3: Tae OS hssignment aeadline is approdching.
  Child 4: Tae iS hssignment aeadline Os approdching.
  Parent terminating...

To get started for this problem, refer to the producer and consumer shared memory example codes on Blackboard for Lecture 4.