COMP 304 - Operating Systems: Assignment 1

Due: Midnight March 6, 2022

Didem Unat Spring 2022

Notes: This is an individual assignment. All forms 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 5% of your total grade. Max possible grade: 75 points.

Contact TA: Ismayil Ismayilov (ENG 230)
iismayilov21@ku.edu.tr

Problem 1

(5x3 points) In Unix/Linux, new processes are created using the **fork()** system call. Calling **fork()** creates a 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()** n times consecutively where n is an integer parameter provided as a command-line argument. 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). **Run the program for n equal to 2, 3, and 5. Provide source code and screenshots showing program invocation and the produced output.** Sample output for $n = 3$ is shown below.

```
Main 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
```

- **Part (b)** As mentioned previously, the child process is the copy of the parent process when created with the **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

command_name where *command_name* is the name of the command to be executed provided as a command-line argument. 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 executing command_name* (For example, if *command_name* is **ls**, this should print *Child finished executing ls.*
Run the program with *command_name* **equal to *pwd* (which prints the current working directory) and *ps f* (which displays a process tree). Provide source code and screenshots showing program invocation and the produced output.**

- **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 6 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 in your solution.**

Problem 2

(10 + 10 points) In this question, you will implement a simple car wash pipeline using ordinary Unix pipes.

- **Part (a)** The process of getting a car washed consists of 3 steps: washing the windows, washing the interior, and washing the wheels. Your program will do the following
  - The parent process (A) is responsible for starting the washing process and maintaining the current car number. The initial car number is 1. The parent process (A) should print "Started washing car <car_num>" where car_num is the current car number. The parent process (A) then creates a child process (B) and sends it the current car number using ordinary pipes.

  - The child process (B) receives the current car number from parent (A) after which it sleeps for 4 seconds. It should then print "Washing the windows of car <car_num>". The child process (B) then creates a child process (C) and sends it the current car number using ordinary pipes.
  - The child process (C) receives the current car number from parent (B) after which it sleeps for 4 seconds. It should then print "Washing the interior of car <car_num>". The child process (C) then creates a child process (D) and sends it the current car number using ordinary pipes.
  - The child process (D) receives the current car number from parent (C) after which it sleeps for 4 seconds. It should then print "Washing the wheels of car <car_num>". The child process (D) then sends the current car number back to the parent process (A)
  - The parent process (A) receives the current car number from child process (D). It then prints "Finished washing car <car_num>"
  - The parent process (A) terminates

- **Part (b)** Your goal for this part is to make the previous program loop n times and wash n cars where n is an integer provided as a command-line argument. The following additions will need to be made
  - After the parent process (A) receives the car number from child process (D), it should increment the current car number and repeat the same steps with the new car number.
  - Once the parent process (A) receives the car number n from child process (D) (meaning that n cars have been washed), it should terminate.

  Run the program with $n = 3$ and $n = 5$. Provide source code and screenshots showing program invocation and output.

You may need to use **sleep(), kill() and fork()** system calls for this problem. You are required to submit your .c source code file along with a snapshot of the execution in terminal for each part.

Problem 3

(15 points) In this question, you will write a C program that determines how long it takes for a *command_name* to run where *command_name* is the name of the command to be executed provided as a command-line argument. To do this, you should do the following:

- Fork a child process which will execute *command_name*
- Child process records the current time before it starts execution
- Parent process waits for the child process to finish execution

- Parent records the ending time once the child process terminates
- Since the parent and child processes are separate, pipes will need to be used to share the starting time between them.
- The parent process outputs the elapsed time

Run the program with *command_name* **equal to** ***history*** **and** ***top*****. Provide source code and screenshots showing program invocation along with the time it took to execute the programs.**
You may need to use **fork(), exec(), wait(), gettimeofday()** system calls and **struct timeval** objects for this problem. *Hint*: a pointer to a **struct timeval** can be written to (and read from) a pipe.

Problem 4

(25 points) In this question, you will learn how to create a kernel module and load it into the Linux kernel. Note that you need to be a superuser on the computer for this assignment.

Background Info:

- *Kernel module*: A kernel module is a code that can be loaded and unloaded into the OS kernel as needed. Using a kernel module, you can extend the functionality of the OS kernel without having to modify the kernel's source code, re-compile, and re-install it.
- *Character device file*: A character device file is a file visible in the file system that serves as an interface to a device driver. It provides a direct, unbuffered access to the hardware device controlled by the device driver. For example, if a user program wants to read or write data into a peripheral hardware device like a USB stick, it will have to read or write the data into the device file in the file system that corresponds to the driver of the USB stick.

Instructions: Design a kernel module that creates a character device file when the module is installed. In this assignment, your device file will not connect to a driver that controls a peripheral hardware device. However, it will be an interface to your kernel module, and the kernel module will serve as a driver that controls a block of memory region allocated for the OS kernel.
The device file that you create should support some file operations that can be performed by user applications on it. A user application should be able to perform the following file operations.

- *open*: Upon opening the character device file, a single large block of memory region is dynamically allocated in the kernel space.
- *release*: When the character device file is closed, the dynamically allocated memory region is freed.

- *read*: When the character device file is read, the content of the memory region is copied to user space, i.e. to the memory of the user application.
- *write*: Upon writing to the character device file, a number of bytes are copied from the application's memory to the dynamically allocated memory region in the kernel space.

For this problem, you are provided with a README file, a template code of the Linux kernel module, a makefile, and a test user application. In the template code, you are only expected to modify the parts that are marked with TODO comments. After making the necessary modifications, you can compile and install the module by following the instructions in the README file. After installing the kernel module, you can compile and run the test application to try to write and read data from the character device.

Some Useful References:

- Info about Linux kernel module: `https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html`
- Info about character device file: `https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html`
- Info about dynamic memory allocation in kernel space: `https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html`
- Info about memory copy between user space and kernel space: `https://developer.ibm.com/articles/l-kernel-memory-access/`

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 the code. You may include a makefile if you like.
   (c) **Do NOT submit object files or executables** to Blackboard.
   (d) Use the problem number to name your source files. For example, p2a refers to the source code solution for problem 2 part a.
   (e) Improper naming or file organizations will result in **5 point penalty** in the assignment grade.
   (f) Selected submissions may be invited for a demos session.

GOOD LUCK