COMP 304 - Operating Systems: Assignment 1

Due: 23:59 April 22nd, 2024

Hakan Ayral Spring 2024

Notes: This is an individual assignment. All forms of cheating will be punished! You may discuss the problems with your peers, but the submitted work must be your own. No late assignment will be accepted. Submit your answers through blackboard. This assignment is worth 5% of your total grade. Max possible grade: 50 points.

Contact TAs: GÖRKAY AYDEMIR, ILYAS TURIMBETOV, MOHAMMAD KEFAH TAHA ISSA

GitHub submissions: You must create a GitHub repository for the project and add the TAs as contributor. Add a reference to this repo in your report or README. We will be checking the commits throughout the course of the assignment and during evaluation.

Problem 1: Forks (10pts: 5 + 5)

In Unix/Linux, new processes can be created using the **fork()** system call. Calling **fork()** creates a copy of calling process and the new process starts executing immediately. After the **fork()** call, both parent and child processes start executing the same code.

Part (a)

Write a C program that 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)

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.

- Use the **sleep()** call for the time requirement
- Run the program in the background (using the & feature of your shell) and then run the command $ps - l$ to see if the child has become a zombie.
- Kill the parent with the *kill* command if you create too many zombie processes.
- Provide the source code and screenshots of the *ps* command.

Problem 2: Pipes (20 points)

Rather than duplicating the parent process, we can create a new process for the children to execute using the **exec()** family of calls, which the parent can monitor. In this part you will write a simple benchmarking program that measures how long programs take to execute.

- Write a C program that forks **n** child processes *simultaneously*, each executing a given **command_name** concurrently. Both **n** and **command_name** are taken from command-line arguments, as shown below. Note that you may need to fork again inside the children.
- The child processes should record the starting and finishing times of the given command and send an estimate of the time it took to execute back to the parent process using pipes. The parent then prints the execution times it receives from the children as they are sent. You can use the **gettimeofday()** library call for this purpose.

```
$ ./bench 5 ls -la
Child 1 Executed in 5.38 millis
Child 5 Executed in 5.62 millis
Child 2 Executed in 5.95 millis
Child 3 Executed in 6.27 millis
Child 4 Executed in 7.20 millis

Max: 7.20 millis
Min: 5.38 millis
Average: 6.08 millis
```

- The child processes should not print anything to stdout or stderr. You can forward those streams to `/dev/null` to suppress the output.
- After all children finish executing, the parent should print the maximum and minimum times of all runs, as well the average time of all children.

Problem 3: Shared Memory (20 pts)

In this part you will be implementing a program to search for a given number in a sequence using multiple processes.

- The parent process reads a newline-delimited sequence of at most 1000 numbers from stdin and parses them into an array.
- The parent creates **n** child processes that go through disjoint portions of the array to search a given number **x**. Both **n** and **x** are provided as command-line arguments. For example, the program should search for the number 98 with 3 children if invoked as follows: `$ ./search 98 3`
- Each forked child must sequentially search for **x** in their portion of the array, and print the index of the element and exit with the return code **0** if found. If a child does not find the given number in its portion, it should instead exit with the return code **1**.
- When any child exits with a success the parent should kill other running children and terminate.
- The last child might have to go through more numbers if the sequence length isn't perfectly divisible by the number of children. Make sure to handle edge cases!

You can use the **shuf** command from coreutils to feed random numeric sequences to your program as follows:

```
$ shuf -i 1-1000 | ./search 98 3
```

Important Remarks

1. All the programming assignments should be implemented and tested on Linux.
2. You can assume all command-line arguments are correct.
3. 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 subdirectory
   (b) Each subdirectory should contain .c source files, screenshots of sample runs if applicable and a single README.txt file explaining how to compile and run the code. Including a Makefile is not obligatory, but appreciated.
   (c) **Do NOT submit any object files or executables**
   (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.