DUE BEFORE: **By the end of your lab session**

CSE-381: Systems 2

Exercise #3

Max Points: 20

Prior to proceeding with the exercise, you should first save/rename this document using the naming convention `MUid_Exercise3.docx` (example: `raodm_Exercise3.docx`).

Objective: The objective of this exercise is to:

- Trace process hierarchy in GNU/Linux
- Learn to create a new process using the `fork` system call.

Submission: Once you have completed this exercise, upload:

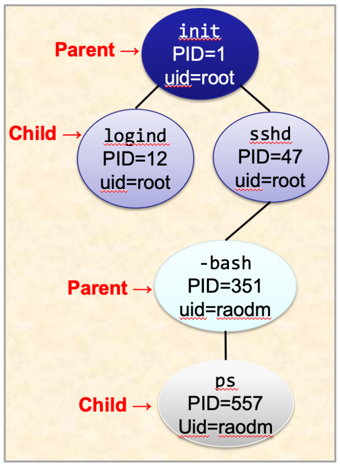
- This MS-Word document saved as a PDF file with the naming convention **`MUid_Exercise3.pdf`**.

You may discuss the questions with your friends, TAs, or your instructor.

Name: **RaoBot**

To help you recap the concepts, watch the video https://youtu.be/7SZdWV1fLxA (at 1.5×) to review some of the concepts associated with the `fork` and `exec` system calls used in this exercise.

Part #1: Recap process hierarchies in Linux



Background: In Linux processes are organized in a hierarchical manner. There is a parent→child relationship between processes, as shown in the adjacent figure. The root of the process hierarchy is `/sbin/init` (synonymous to the kernel). Each process has a unique PID (process ID). The process ID of its parent is called PPID. When processes are listed by the `ps -fe` command, it shows the PID and PPID values. Using this pair of IDs, the lineage of a process can be recursively traced back to the `init` process, at the root of the process tree. This is what we will be doing manually in this part of the exercise. **Homework #3 will focus on automating tracing the process hierarchy**.

Exercise:

Complete this part of the exercise via the following procedure:

1. Open a PowerShell/Terminal and SSH into `os1.csi.miamioh.edu`.
2. At the bash shell prompt run the command `ps -fe` to obtain a snapshot of the list of processes running on the server.
3. Next, starting with your `ps -fe` command entry (in the output from previous step) iteratively use the `PPID` value corresponding to each process and locate the process ID corresponding to it. In other words, recursively trace the process hierarchy tree using the `PID` and `PPID` values until the `PPID` value is 0 (zero) corresponding to the `/sbin/init` start-up kernel process. List the sequence of processes you traverse in the table below (add more rows to the table as needed):

PID	***PPID***	***CMD***
18877	18601	ps -fe
18601	18600	-bash
18600	18436	sshd: raodm@pts/2
18436	678	sshd: raodm [priv]
678	1	/usr/sbin/sshd -D
1	0	/sbin/init

Note: Ensure you understand what you are doing here because you will be expected to automate this recursive tracing process in the next homework.

Part #2: Explore Virtual Memory layout after the `fork()` system call

Background: The `fork` system call provides a mechanism to clone a running process. In Unix and Linux, `fork` is the mechanism that is used to create new processes and run other programs. When a process forks a new child process is created. Both parent and child processes are identical copies of each other and resume operation right after the call to `fork()`. The only difference is that the return value of `fork` is different in the parent and child. The child and parent have exactly the same virtual memory layout. That is variables and data structures all have same virtual memory address. However, they are mapped to different physical addresses thereby making them different processes! Review this concept if needed from previous week's lecture video (at time 12:18) - https://youtu.be/PPSmAlnrJxo?t=1278.

Exercise: Complete this part of the exercise via the following procedure:

1. Download and SCP the `ex3_1.cpp` to the server.
2. Log onto `os1.csi.miamioh.edu`
3. Briefly study to the `main` method. Notice how `fork` system call is used to create a child process. Notice how the address-of (`&`) the variable `retVal` is printed in both the parent and child processes.

4. Next, compile and run the supplied program from the bash shell prompt via:

```
$ g++ -g ex3_1.cpp -o ex3_1
$ ./ex3_1
```

5. Copy-paste the output from the program in the space below:

```
In child process (pid= 93843)
  retVal (@address: 0x7ffee18a09f8) = 0
In parent process (pid= 93842)
  retVal (@address: 0x7ffee18a09f8) = 93843
```

6. From the above output complete the following information:

Address of `retVal` variable in parent process:	`0x7ffee18a09f8`
Value of `retVal` variable in parent process:	`0x7ffee18a09f8`
Address of `retVal` variable in child process:	0
Value of `retVal` variable in child process:	93843

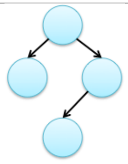
7. In the above information, you will note that `retVal` variable has the same memory address in both processes but it has different values! What? How is that possible? How can two variables in two different processes have exactly the same memory address (*i.e.*, seemly they are stored in the same location in memory) but store completely different values (review lecture video from previous week: if needed to answer this question https://youtu.be/PPSmAlnrJxo?t=1278)?

There is nothing special/new here. It is normal because all processes run in "virtual memory" and have the same virtual memory addresses. However, the OS maps these virtual memory addresses to different physical memory addresses where different values can be stored. So even though the virtual addresses are exactly the same on the 2 (or more) processes, they are all mapped to different physical memory addresses by the OS and hence can store different values. Typically, each program has exactly the same virtual memory layout when running, but each process is mapped to different physical memory addressees by the OS

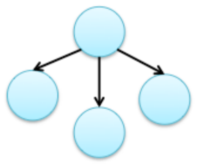
Part #3: Practice visualizing & understanding operation of fork

Background: Recollect that `fork` system call creates a clone of the exiting process and both parent and child continue running right after the `fork` system call returns. Using the return value of fork (*i.e.*, zero in child, and non-zero in parent), different process hierarchies can be constructed (using if-else statements appropriately).

Convert the following process trees to corresponding C++ source code -- that is, the C++ source is implemented to create the process hierarchy shown in the figure. The first question has been completed to illustrate an example.

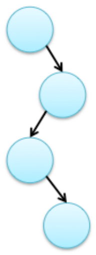


```
int main() {
    fork(); // First fork only in parent
    fork(); // Parent and child both fork
    return 0;
}
```



```
int main() {
    // Uses short-circuiting in C++/Java
    bool done = ((fork() != 0) && (fork() != 0) &&
                 (fork() != 0));
}

// Alternative implementation (same as the above program
but expanding the conditions using equivalent if-
statements)
int main() {
    // Same logic as above but expanded with if-else
    if (fork() != 0) {
        if (fork() != 0) {
            fork();
        }
    }
}
```



```
int main() {
      if (fork() == 0) {
            if (fork() == 0) {
                  fork();
            }
      }
}

// Alternative implementation (same as the above program
but condensing the if-statements using shortcircuiting)
int main() {
    // Same logic as above but using short-circuiting
    bool done = ((fork() == 0) && (fork() == 0) &&
                 (fork() == 0));
}
```

Part #4: Understanding the basics of **execvp** method

Estimated time: 20 minutes

Background: The task of running a completely different program is performed by the `execvp` system call (The `execvp` system call is one in a family of `exec` system calls). The `execvp` system call requires the following 2 arguments:

1. The first argument is the name or path of the program to be executed
2. An array of C-strings for command-line arguments. Note: As per convention, the 1st command-line argument should always be exactly the same as the name/path of the program being executed.

Exercise: In this part of the exercise, we are going to build some strength on the basic understanding of the `execvp` system call:

1. Why does the `execvp` system call work with addresses (or pointers)?

   The OS works with the most common subset of information that is programming-language agnostic. Such information is memory-address or pointers. Consequently, the `execvp` system call works with pointers.

2. How does `execvp` system call determine when the array of command-line arguments (*i.e.*, the 2nd argument to `execvp`) have ended?

   The last entry in the array of command-line arguments should be a `nullptr` (or memory address `0x0`). The `execvp` system call uses this information to determine when the list of command-line arguments are ending.

3. What should be the first command-line argument passed as part of `execvp` system call? why?

   The first command-line argument should always be exactly the same as the command being executed. This is the convention.

4. How does the OS determine the characters constituting a given command-line argument?

   The OS uses a programming-language agnostic approach for string processing. In this case it uses a sentinel value of `'\0'` to logically terminate each string. Note that in C/C++ the trailing `'\0'` is automatically added to strings.

5. Upon successfully running a given program, what is the return value of the `exec` system call?

   The `execl` and `execvp` systems calls do not ever return after successfully starting to run another program. That is because our program is no longer running and is replaced with the newly start program. In other words, our program ceases to exist. Hence, there is nowhere for `execvp` or `execl` to return to.

6. There are several types of exec system calls. The common ones are `execl` (where the exact number of command-line arguments are known) and `execvp` (with variable parameters, *i.e.*, number of parameters are not known). Let's practice a few calls to the `execl` syscall first.

   Complete calls to the `execl` system call (**Notes**: Yes, the command-needs to be repeated because the 1st command-line argument is always the program being run. Note that **`nullptr`** at end is important):

Description	**C++ code fragment (1-liners each)**
List all files in your home directory	`execl("ls", "ls", "-l", "~/", nullptr);`
Show list of all processes running on the machine	`execl("ps", "ps", "-fe", nullptr);`
Kill process with `pid` 1234	`execl({"kill", "-9", "1234"});`
Run program named `ex4` in PWD.	`execl({"./ex4"});`

7. The `execvp` syscall can be a bit tricky to work with. Hence, we will use a helper method. What was the signature (name and parameters) of the helper method that was suggested to ease using the `execvp` system call (Hint: see: 16 minutes into the video https://youtu.be/7SZdWV1fLxA?t=960)? What was the caution discussed about the helper method?

   The suggested helper method is `myExec(std::vector<std::string> args);` The caution discussed was that this method is designed to take a mutable list of command-line arguments so that they can be passed to the OS via the `execvp` system call. It is best not to modify this method but just copy-paste and use it.

8. Let's ensure we are paying attention to how to work with the `myExec` method by completing the method calls below (the first one is already completed to illustrate an example. Assume you are given the following `myExec` method that runs the specified program –

```
using StrVec = std::vector<std::string>;
int myExec(const StrVec& args);
```

Description	**C++ code fragment (1-liners each)**
List all files in your home directory	`myExec({"ls", "-l", "~/"});`
List all files in the directory `/usr`	`myExec({"ls", "-l", "/usr"});`
Using cat command, print the contents of the file `/usr/share/common/gpl.txt`	`myExec ({"cat", "/usr/share/common/gpl"});`
Kill process with `pid` 1234	`myExec({"kill", "-9", "1234"});`
Show list of all processes running on the machine	`myExec({"ps", "-fe"});`
Run program named `ex4` in PWD.	`myExec({"./ex4"});`

Submit files to Canvas

Upload just the following files to Canvas (no, we are not using the CODE plug-in for this exercise):

- Upload this document (duly filled with the necessary information) saved as a PDF file using the naming convention `MUid_Exercise3.pdf`.

Upload each file individually onto Canvas. Do not upload zip/7zip/tar/gz or other archive file formats for your submission.