Process Management Sim Report

Implementation Summary

The way my lab works is as follows. The commands that will be used within the exec statements are written beforehand as a 2D char array. Each of the 10 elements within the first layer of the array are either a three or two long char array. This contains char strings for each of the commands, followed by any sort of additional field they may need such as a file path and ended with a null pointer. Following the setup, a for loop runs 10 times. In each loop, the fork function is called splitting the process into a parent and child. From there the child process will print its PID, its parent's PID, and the name of the command this child will be executing. The name of the command is snagged from the command array set up earlier. The index of the command being retrieved matches the iteration the for loop is currently on. After the print statements, the entire char array of the designated command is placed into an EXECVP call.

The command that will be run is dependent on what is listed in the command array; however, due to the nature of execvp calls, the process of the child will switch from running lab2.c to running the specified command. Importantly, all of the code that was running before in the C file is no longer running. The child will simply perform the command and either exit as normal or encounter an error before I can get off the ground. In the event of an error, we are still within the C file and simply print an error message and manually terminate the process ourselves. Therefore, the child process will never actually make it to the next iteration of the for loop. Only the parent will. This is also what prevents the child process from forking off children of its own. Because all children processes will have switched to a command instead, they will not survive long enough to fork. This is how the usual pattern of forking off children in 2^n power is limited to 10 children only.

Then the next iteration of the loop begins again in the parent process. Due to the sharing of all memory up to the point of a forking, the state of the parent’s iterator is preserved into its new child. This allows it to retrieve the correct command from the array and not simply call the first command 10 times in a row. After the loop ends and all 10 children have attempted to run their EXECVP commands, the parent process has to wait for all of its children to exit or terminate. The parent uses the waitpid() function to halt the parent process until a child is detected as ending. The parent does not know which of the children this is, so it uses a for loop to wait 10 times, covering each child. An integer status is declared to store the exit state of the recently ended child, either exited, terminated, or missing. Based on this status, a proper ending message is printed respectively. After all children are done, the parent can return and end the program.

Child Process Execution

The system calls I have decided to use for this lab via execvp are as follows. One child will print my name into the terminal using the echo command as stated in our lab report. The second will use ls to list all files within the current directory. The third will use the date command to return the current time and date from the system clock. The fourth runs the whoami command, which retrieves the username from userdata. The fifth and sixth processes will print out the PIDs/commands of all currently running processes and the path of the currently working directory, respectively, with ps and pwd. The seventh command is cat, which prints out into the terminal any file I choose, in this case, the lab itself. Process 8 uses mkdir to create a new directory which I have named homework, and process 9 uses the touch command to create a new file also named touch.txt. (Note: touch.txt is not created in the homework folder, as I did not know which would run first due to the varied nature of process scheduling). Last, the 10th command is ping, which pings the IP of a given web domain—in this case, my GitHub profile.

Process Creation/Management and Parent-Child Interaction

In operating systems, a process is just a specific program that is being currently run. This could be a program hardcoded into the operating system or something user-written. The process is made out of one or more threads of instructions being interpreted and executed. Each process has a specific ID so that they can be tracked, which is utilized in my program. These processes are managed via the Process Control Block (PCB), which is a data structure used to hold all of the needed info on a specific process. A computer is often trying to run many processes at the same time, so the PCB is able to track and manage what processes receive what memory and what set of instructions are scheduled to execute for how long.

Within this process management simulator, forking is specifically used to create multiple processes. The program begins with a parent process, but as it continues to execute, the fork() call is utilized to create an additional process, referred to as a child, to act as a duplicate of the parent. The parent and child do not share memory or an ID with each other. Child processes are given their own IDs, and on creation, they receive an exact copy of the parent's memory. However, this memory is stored in a different location, and any changes made to the memory of either child or parent will not affect the other.

The parent and child processes interact in the following manner:

1. The parent creates child processes using fork().
2. Each child executes a system command via execvp().
3. After execution, the child process terminates.
4. The parent process waits for all child processes to terminate and tracks their status using waitpid().

This process ensures that the parent manages the lifecycle of its child processes, ensuring that they execute their tasks (commands) and terminate properly. The parent waits for the children to finish their respective tasks and reports their status before concluding the program.

In this demo, I use forks to run 10 child processes. These ten children are used to run a variety of sample execvp execution statements demonstrating various system calls. Normally, a for loop will create children in 2 to the power of N amounts. However, these child processes are being switched from my C program to a system process, which will close themselves before they have a chance to fork on their own. The parent process waits for the child to finish, keeps track of their exit status, and reports all of them, managing the creation and termination of each child.