Overview

Process management is a fundamental component of resource sharing and multitasking in contemporary operating systems. Effective parallel computing is made possible by the capacity to design, carry out, and synchronize numerous processes. This project investigates the use of fork() to create processes, execvp() to execute various commands in child processes, and waitpid() to synchronize. This project shows how a parent process can create multiple children, carry out distinct tasks in each, and efficiently manage their lifecycle through a real-world implementation in C.

Summary of Implementation

The C program defines an array of shell command arguments for every child and starts with the required header inclusions. The parent process uses a loop to use fork() to create ten child processes. The return value of fork() determines whether the current process is a child or the parent.

Then, using the execvp() function, each child process swaps out its image for a distinct shell command, like ls, whoami, or echo "Hello + Aaron." The child exits with failure and an error is printed if the execvp() call is unsuccessful.

The PIDs of the generated child processes are stored in an array in the parent process. The parent uses waitpid() to wait for each child to finish after they are all created. The parent then reports the exit status or signal number in accordance with whether the child ended normally or as a result of a signal.

The program is compiled using a Makefile, which streamlines and automates the build process.

Findings and Remarks

A. Development and Administration of Processes
Every time fork() was called, a new child process was successfully created.

The program structure clearly separated task responsibilities because the return value of fork() makes a clear distinction between the parent and child logic blocks. Even with ten child processes, this kept things manageable and readable.

The project illustrated how execvp() enables the child process to change into a new process image by giving each child a distinct command. The new executable replaces the child's original

process code once execvp() is called, guaranteeing that no code is executed after execvp() unless an error occurs.

B. Communication between Parents and Children
Waitpid() was used by the parent process to synchronize with each child. This function gives specific information about how a child ended in addition to blocking until that child does. This allowed the parent to retrieve the appropriate exit code or signal number and print whether a child exited normally or was terminated by a signal.

Every child's output verified that distinct tasks were completed, and every child reported their PID and the command that was executed. Completion reports were sent to the parent in the order that the child processes ended, not necessarily the order in which they were generated.

In conclusion

This project effectively illustrates basic UNIX process operations. Fork() was used to create several separate processes. Waitpid() offered dependable synchronization and child termination monitoring, while execvp() enabled kids to run a variety of meaningful commands. This project lays a strong foundation for future research into concurrent programming in C by emphasizing the significance of process control, error handling, and synchronization in systems programming.