# Introduction

Process management is a crucial aspect of modern operating systems, allowing for multitasking and efficient resource utilization. This project aims to implement a process management simulator using C, demonstrating the use of fork(), execvp(), and waitpid(). By creating and managing multiple child processes, the program showcases fundamental principles of process execution, synchronization, and termination.

# Implementation Summary

The C program is structured to create exactly ten child processes, each executing a specific command from a predefined list. The program utilizes key system calls:

* fork(): To create new processes.
* execvp(): To replace the child process's memory space with a new program.
* wait(): To synchronize the parent process with its child processes.

The program defines a set of commands stored in a 2D array, ensuring that each child process executes one command. If there are more child processes than commands, the commands are cycled through using modulo arithmetic.

## Key Components:

* **Constants:** NUM\_CHILDREN (10) and MAX\_ARGS (3) control the number of processes and command arguments.
* **Command List:** A static array of commands with arguments.
* **Process Loop:** A for loop with fork() to create child processes.
* **Execution:** execvp() to run commands in child processes.
* **Synchronization:** A while loop with wait() to handle process completion.

# Results and Observations

## Process Creation and Management

The program creates processes using the fork() system call, which duplicates the parent process, resulting in a child process with a unique Process ID (PID). The child determines which command to execute based on its iteration index and cycles through the command list if necessary.

* **Process Identification:** The pid variable distinguishes between parent and child processes.
* **Error Handling:** The program checks for fork() and execvp() failures, using perror() for debugging.

## Parent-Child Interaction

The parent process coordinates the lifecycle of child processes:

* **Creation:** The parent initiates each child process through fork().
* **Execution:** The child process immediately replaces its memory space with the specified command via execvp().
* **Waiting:** The parent waits for each child to terminate using wait(), which returns the PID of the terminated child and its exit status. This ensures that the parent process does not exit before all children complete.

Interaction is unidirectional in this case, with the parent creating and monitoring, while the children execute commands and terminate.

# Conclusion

This implementation demonstrates core concepts of process management in UNIX-like systems, highlighting the creation, execution, and synchronization of multiple processes. The program effectively uses fork(), execvp(), and wait() to manage child processes, ensuring orderly execution and completion tracking. The structured approach aids in understanding the interaction between parent and child processes, error handling, and process lifecycle management.