Introduction: Process management is the process by which an operating system creates, controls, and terminates processes. A process is an active instance of a computer program currently being executed by one or more threads. Process management ensures that each process receives CPU time, memory, and resources while also allowing processes to communicate and synchronize. This process is important because it enables multitasking and reliable execution of multiple programs at the same time. In Unix systems process management is mostly handled through system calls such as fork(), execvp(), and waitpid().

Implementation: My overall program is designed to create 15 unique child processes using a loop that calls fork() for each child. The parent process stores the PID of each child in an array called childPids[], maintaining the order in which the children were created. After creating all children, the parent waits for each child to finish using waitpid() in the same order that the children were created.

Results and Observations: My results and observations were that during the execution the parent prints the PID of each created child in order. For example, "[Parent] created Child #0 with PID = 16324". Another observation I saw was that each child prints its own PID and parent PID before executing its assigned command. For example, "[Child #0] PID=16324 PID=16323 about to run: ls". This tells me that each child was correctly given a unique PID and that the parent maintains the correct creation order in the childPids[] array.

The program also demonstrates process creation and termination order, for example the parent creates children sequentially, but the operating system schedules each child independently. This causes some children to finish before others, even if they are created later. For example, a child that ran the sleep command could terminate after other children have already finished. But the parent still waits for each child in creation order using waitpid(), which would ensure that each child's termination is in the same order they were created.

I also observed how the program correctly detected normal exit vs signal termination; the parent would determine how each child terminated. For example, WIFEXITED(status) would return true if the child exited normally. WEXITSTATUS(status) retrieves the exit code, WIFSIGNALED(status) returns true if the child was terminated by a signal, and WTERMSIG(status) retrieves the signal number.

Conclusion: This project gave me some hands-on experience on how Unix process management works, by using fork(), execvp(), and waitpid(). The program demonstrated how the operating system creates child processes, replaces code and memory space, and reports the termination status. It also showed me the difference between creation order

and termination status and how signals can cause odd terminations. Overall, this lab showed the importance of process control in operating systems and demonstrated some of the tools for process management in Unix.