COSC 315 - Closh Program - Readme

Tanvir Kang - 35470160
Carson Perreaux - 85322311
Zachary Maludzinski - 32950164

Preface

Our approach to this project consisted of treating it as three distinct parts: the timeout, sequential execution and parallel execution. This allowed us to develop each of the parts as distinct functions while also dividing the work required between the three group members evenly. The functions we decided that would be most logic were first for the serial execution, the parallel execution, and for both a timer functionality that can commonly work for both.

Testing File

The testing file used in this case is called “testing”. It is a simple program that prints “Hello World!”, then 2 seconds later prints “hello”. The testing method for the sequential was that if the program prints “Hello World!” “Hello” the specified number of times was successful. For the parallel function, we would expect “Hello World! Hello World! …”, “hello hello …” for the specified count. The timer functionality was tested by using “gedit” on Linux, if the program remained open for longer than the specified timeout it was terminated, and the next sequential one would open.

Void sequentialExec(int count, char** cmdTokens, int timeout)

Count indicates the number of processes to initialize and execute. cmdTokens indicates which process to run along with any other additional arguments. Forks a child thread within a for loop and uses execvp to execute the new process using the child. The parent thread waits until the child process is complete before iterating through the loop again and beginning a new process. Printf statements are used liberally to indicate to the user what is currently happening with the system and aid in determining error sources.

Function written by Zachary Maludzinski.

Void parallelExec(int count, char** cmdTokens)

Parallel Execution is a recursive void function that is called from main when the user inputs ‘p’ to indicate parallel execution. The inputs are count, which indicates the number of parallel instances required by the user. The cmdTokens are a char array that contains the name of the program called, and its arguments. The logic is, if the counter variable is greater than 0 (1 or more), a child will be forked, the child will again call the parallelExec function with the count variable decremented by 1. As calls are made, the children are all pushed to the recursive stack. When the condition count > 0 is no longer satisfied the execution of all children begins. The Nth child begins along side N-1, N-2, to N-Count children all begin to execute at the same time. When the count > 0 condition is no longer satisfied the function call return, going back into

the main, where the statement wait(NULL), indicates that the entire process must stop here and wait for all children to finish executing.

This function is called from the main, when parallel==TRUE. Upon entering the function there is an immediate fork creating a child and parent, the child calls parallelExec, whereas the parent calls the timeOut function. If the timeout occurs before the function call is complete it will be killed. However if the function returns before the timeout is complete, the function will call kill upon itself and end the function.

Function written by Tanvir Kang.

Void timeOut(int childId, int timeout)

A void function that accepts a child ID and timeout arguments as parameters. If the timeout is greater than 0 the function will wait for the specified period of time, after the time has elapsed it will kill the process with the specified child ID using kill with SIGKILL. If there is a 0 value provided, the function will wait indefinitely until the specified process ID finishes execution. This function is used within the two execution methods.

Function written by Zachary Maludzinski, and Carson Perreaux.