COSC 315: Operating Systems - Project 1

February 07 2022
Riley Clark - Main and Timeout Sequences
Reid Folk - Documentation
Aalia Omarali - Timeout Sequences

For this assignment, our team was tasked with designing and implementing a functional clone shell in C. To accomplish this, we utilised system calls such as fork() and exec() to take the input name of the program, the number of processes the program should run, whether the processes should run sequentially or in parallel, and a timeout specifying the maximum duration of each process. Our code can be broken down into 4 major parts: The Main Parallel Sequence, The Main Sequential Sequence, The Parallel Timeout Sequence, and The Sequential Timeout Sequence.

Main Parallel - main(), if(parallel)

In main, just below the parsing code, we first initialise an array to keep track of the forked pids for parallel processes. Then, if our process is parallel, we run a for loop from 0 to the count specified in the parse.

At the start of the for loop, we first fork the pid so that the child can call execvp. If the pid is 0, we print the pid and it's parent to the console, empty the child's buffer, then execute the specified command. If an error occurs, we print to the console that the command doesn't work, then exit the loop. Similarly, if the pid is less than 0, we print to the console that the process can not be created and exit the loop.

Finally, once the for loop has completed, we send the specified timeout, pids, and count variables to our parallel timeout handler.

Main Sequential - main(), else

In main, just below the parsing code, we first initialise a variable to hold our pid for sequential processes. Then if our process is sequential, we run a for loop from 0 to the count specified in the parse.

At the start of the for loop, we follow the exact same procedure outlined in our parallel sequence, however before the end of each loop, if the pid is greater than 0 then the parent process will wait until the child process is done by sending the specified timeout and pid to the sequential timeout handler.

Parallel Timeout - timeoutHandlerParallel()

This handler, which is called at the end of the main parallel sequence, takes the time in seconds, and deletes the process if is has been running longer than the time specified in the timeout variable.
The handler first initialises 5 variables, stat, before, after, difference, and done. Then, it runs a while loop which ends when difference is less than or equal to timeout. At the start of the loop, after is set to time(null), and difference is calculated by the difference between before and after. Once this is set, a for loop is run from 0 to count.

At the start of this for loop, it first checks whether or not the current pid is equal to 1000000. If it is, the loop is exited, but if it isn't we get the current status of the child and check if the process is complete. If it is, then we let the child terminate, set the pid of the child to 1000000, and if all processes are complete we return to the normal flow.

Once we have exited the initial while loop, it is then time to kill processes that are incomplete using SIGKILL, and print to the console that the specified process has been terminated due to timeout.

SequentialTimeout - timeoutHandlerSequential()

This handler, which is called at the end of the main sequential sequence, runs in a similar fashion to the parallel timeout handler. However, once after and difference have been set in the while loop, we get the status of the child immediately, and check if the child process has completed. If the child process has not completed, then we kill that process and print to the console that it has been timed out. If the child process has completed, then we return back to the main sequence.