CMPUT 379 Assignment A1
Christian Lo, 1498360

Objectives:

The objective of this programming assignment was to learn about using various UNIX calls in C++/C regarding processes and process control, including fork(), waitpid(), execvp(), and kill(). Concepts such as pipes are also learned with the use of the popen() and pclose() functions. These concepts are taught through the design and implementation of a homebrew command line and a system process monitor. The command line will use concepts of processes and signals, while the process monitor covers the topics of signals as well as pipes. The value in this assignment is in both in it's introductions to using various UNIX commands, as well as a good review of C/C++ for those who have not used said languages in for a fair period of time.

Design Overview:

There are several aspects to both command line as well as the process monitor.
The design of the command line can be chalked down to a forever running while loop
In each loop:

- A command prompt is printed and user input is waited on.
- The user's input is then broken down into a command, an executable/job number, and arguments for an executable. The command is read, and will either call execvp() will the executable, or do something with a given job number.
- If the command is 'run', the program create a child process (with fork()), which will call execvp() with both the user given executable name as well as any additional parameters the user may have added. The program has a global process counter, so the 'run' command will be rejected if the command is not formed properly or the process counter reaches the maximum amount of processes.
  The parent process (the command line) will check if the child properly executed command it was given by checking if the child process terminated (By calling waitpid(...,&status,...) and looking at the resulting status of the child) with an exit code of 1, indicating an error. If the child process does not return an error the parent process will increment the job counter, indicating that a valid process was created.This is a slight deviation from what is expected, as it was posted on the forums that incrementing the job counter should occur if the fork() command executes successfully, while this implementation increments if fork() and execvp() successfully execute. This implementation should be better as it doesn't add invalid jobs to the jobs list, thus avoiding a whole suite of bugs. As specified there is no way for this counter to go down. Any successful child process gets added to the jobs list, which can be referenced by other commands. Each process in this list is referred to as a job, and the index of the job in this array is considered to be a the job ID, or jID for short. The job array is of type job_t, where each job_t can hold a command name as well as the pid of said command.
- If the command is 'list', the command line will "ping" each of the child to make sure they exist and not terminated with the kill(pid,0) function. Any process that has not been terminated will be printed out to the terminal.
- The command 'suspend [job id]' will first reference the job in the array, then checks if the process exists with kill(pid,0), then sends a suspend signal to the given process after it has verified that the input is legitimate. The signal sent will be SIGSTOP via kill() to stop the process.

- The command ‘continue [job id]’ will perform the same operation as ‘suspend’, only with it sending the SIGCONT signal to resume a process.
- The command ‘terminate [job id]’ again operates similarly to ‘suspend’ and ‘continue’, with slight adjustments, some according to spec, others going above it.
  A signal of SIGTERM will be sent, however, a waitpid() function will be called to reap the zombie process. A feature added is that calling ‘terminate’ will resume any stopped process before terminating it, as there is not much good in sending a terminate signal to a stopped process. This does not violate the specification of the assignment, and arguably makes the end result a better product.
- The command ‘exit’ will terminate any jobs in the jobs list, and reap the process with waitpid(). The command will then break out of the program loop, ending it.
- The command ‘quit’ also breaks the while loop, though it does not kill and child processes.
- The ‘help’ command sends a synopsis of each command to the user.
- Any other input will prompt the user that the command is unknown.

All other minor details of the project were implemented, such as the setting the r_limit to 10 minutes, as well as getting the times of the processes with the times() command.

The monitor was a more straightforward process, as aside from the initial command arguments given when executed, there is no user input. The commands are parsed, and as specified a pipe to the ‘ps’ command is opened. Each line if read from the file descriptor, printed to the screen, and read into an array of processes, where the command, pid, and ppid are stored from each line to an array of proc_t, or of process type.

Assuming the target process is alive, the monitor will then call a custom function to add any children of the target process to a watchlist, to be terminated if the target process is terminated. The function is recursive, so it will therefore add any descendents of children and so forth if needed.

If the target process is not alive or is not in the monitor for whatever reason does not read the target process from ‘ps’, it will then terminate and wait on any processes in the watchlist (with kill() and waitpid() respectively) before terminating.

Project Status:

The project meets the specifications given in the assignment and moreso. Small quality of life tweaks such as command prompt color and a ‘help’ function have been added, though they do not drastically improve nor alter the project. The project, the command line specifically, works under many fringe cases/inputs, and is free of any bugs within a fairly large given set of inputs.
The most difficult part of the assignment was definitely refreshing on C++, as well as the best practices to use with it. On a high level, one of the most prevalent issues was trying to figure out the man docs, though after a couple of reads through the documentation now makes much more sense.

One of the biggest technical snags was trying to get the command line to increment it’s job count if the command the child process runs is valid. This is mostly because the child process can either: exit after an error in execvp(), successfully run a non looping command like ‘ls’ and terminate, or successfully run a looping command like ‘xeyes’ and not terminate. It took awhile to figure out a consistent solution, since I couldn’t just check if the process was running (because ‘ls’ is valid and it terminates), and I couldn’t get the child to signal to the parent if it successfully ran or is running a command (because executing a command successfully replaces the child process with that command, meaning additional code cannot be added).

Testing and Results:

The code went through several tests to ensure that both programs ran smoothly. For the command line, incorrect inputs were entered, such as “”,”rufdafa”,”____”. All of these either returned an error stating that the command is malformed or in “”’s case, the command line will simply ignore the command and prompt again, similar to that of a bash terminal.
The ‘run’ command was tested extensively, as there were many situations in which a bug could occur. A command such as “run ”, “run dsaasffs”, or “run 12345” will all fork() and attempt to run execvp(), however, the execvp() call is checked if it has an error status, and if so, prints the error with perror(), and the child will exit with a status of 1. The parent listens for this exit code, and does not increment the job counter if it gets an exit status of 1. This results in the program being both verbose and fault-tolerant. The same scenario plays out in the case of invalid parameters, such as “run xeyes -a -invalidParam”, since execvp() will give an error due to the invalid commands. The run command also works with programs that terminate , such as ‘run ls’. This implementation increments the job counter as the command was successfully run. Commands such as ‘run xeyes’ or ‘run xclock’ both work, and behave as specified. Commands such as ‘run ./myclock.sh’ will successfully run programs not on the PATH, so long as they are actually in the specified directory.

Every other command was tested in conjunction with another. Suspending, resuming, and exiting, all work in isolation on a program such as ‘xeyes’, which is the intended use. Further more, incorrect job ids will return an error and attempting to signal a terminated process will return an error. Attempting to terminate a stopped process has been tested, and the programmed result will resume a process before killing it, as sending a terminate signal to a stopped process doesn’t terminate the process unless it has been resumed. This is another mild change to the specification, as it can be assumed that a user wants to completely end a process if they call ‘terminate’, rather than just send a signal for the sake of sending a signal. The list function has also been tested to make sure that a process will be listed unless terminated either by completing (‘ls’) or user termination.

The process monitor has also been tested thoroughly. The process monitor requires a job number parameter and enforces that the input must be a number. The monitor also does the same for the interval value, though the interval value has a default value and does not need to be entered. The monitor has been tested to monitor and terminate when watching a single process, a process with a child, a process with multiple children, and a process with multiple children where one child has a child process of its own. In all cases, the monitor will terminate all descendants of the parent process, whether it is a child process, a child of a child process, etc.

Acknowledgements:

Multiple resources have been used in this assignment. Namely http://www.cplusplus.com/reference/, both textbooks that are used in this course, the man documentation,
https://stackoverflow.com/questions/2808398/easily-measure-elapsed-time,
https://stackoverflow.com/questions/16029324/c-splitting-a-string-into-an-array,
https://stackoverflow.com/questions/20731/how-do-you-clear-a-stringstream-variable,
https://stackoverflow.com/questions/236129/how-do-you-clear-a-stringstream-variable,
as well as help from a TA to help debug setrlimit() and getrlimit().