Please fill in each section of this documentation file with the information which we will need to mark your coursework.

**Full Name:** Barnabas Forgo

**University Username:** psybf

**Computer Science Username:** bxf03u

**Are there any special instructions we need in order to get your programs to run?**

Just use the “-lstdc++” flag when compiling to link the standard C++ library.

**What problems do you currently know about your programs?**

No known problems.

**Requirement 1:**

Comment on how you generated the exact number of child processes, and, if more processes are generated, what causes it

A simple for loop is used. Its max value is set by the #defined constant, NUMBER\_OF\_PROCESSES.

**Requirement 2:**

Comment on how you made the parent process (and only the parent process) wait for the child processes to finish (while all processes can run in parallel)

After all children processes are created the parent exits the for loop and enters a while loop. The loop condition is “-1 != (pid = waitpid(-1, &status, WUNTRACED))”. waitpid(-1,…) means wait for any children process to end, it’s return value is saved to pid, and if that is not -1 (meaning not waiting for anything else) it stays in the loop (waiting for the next process to end).

**Requirement 3:**

General comments: The output shows that the processes were switched randomly, some getting more time than the others. This might be caused by other processes having higher priorities (for example when no process was switched to for a long time). Over several runs though it was apparent that this is just an anomaly; each time a different process got a little bit more or less time on the CPU.

Comment on how the time tracking was implemented, what data structures you used, how synchronisation is guaranteed (if at all required), and how you pass on the base time between the parent and the child processes

A file mapped, shared int array was created in main with a size of NUMBER\_OF\_PROCESSES \* MAX\_EXPERIMENT\_DURATION. Then the run-times were logged with runTimes[diff \* NUMBER\_OF\_PROCESSES + i] = i+1 (diff is the time since start, i is the process index). This in practise means that it is a 2D array where each process has a row and each process logs its number at column diff. When the process wasn’t running the value of the array at that location remained 0. This way no synchronisation is needed; the processes don’t interfere with each other; and the results can be logged in an orderly fashion when the experiment has ended. The start time is copied into the child process’ memory when it’s forked; no extra code needed for that.

The CSV file is generated by piping the program’s stdout to a file.

**Requirement 4:**

General comments: The code showed in the requirements document was used. sched\_setaffinity is called in main since child processes inherit their parent’s affinity. The output clearly shows that on a single core the processes are switched evenly; each getting about the same time on the processor.

**Requirement 5:**

General comments: Again the code was copied from the requirements document. This time though, setpriority was called from inside the child process, since each has to have a different priority. The output was as expected: the process with the higher priority got most of the computing time, but others with a lower priority still got the chance to run.

**Requirement 6:**

General comments: The for loop that spans the child processes was modified to run one more time. Its last run was set to run the ps command. I added the additional -H flag to show the process tree instead of just a confusing list.

**Requirement 7:**

General comments: -

Comment on how you implemented the SVG visualisation, and how, if at all necessary, you had to modify the data structures used by your process to allow you to generate the SVG files, and what types of synchronisation you had to implement, if at all necessary

Since the data generated can be accessed in the parent process it was easy to create the SVG graph. The only change compared to CSV was to calculate the location of the rectangles and print wrappers so the output would be a valid html file. One rectangle represents a single millisecond when a process was running. Each of these rectangles are 1px wide and the whole graph is MAX\_EXPERIMENT\_DURATION wide.

**Requirement 8:**

Based on your experimentation, reflect on how the process scheduler in Linux works, and on the questions asked in the coursework specification

Based on these experiments I am confident that bann uses the Completely Fair Scheduler. It is pre-emptive since it didn’t let any of the processes to finish completely before switching and this wasn’t requested from the OS. This is especially apparent when the processes are ran on a single core with the same priorities. In this case all of the processes get about the same time on the core and they are switched evenly. By definition the CFS algorithm doesn’t cause processes to starve. This is done by only assigning timeframes to each process that is equal for same priority processes (that is, if there are 4 processes each process gets 1/4 of the time). On the other hand when the priorities are set lower than normal they get less time on the CPU, but they still get some on a constant frequency, therefore not starving them.