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

**Full Name:** Natalia Sharon

**University Username:** psyns3

**Computer Science Username:** nxs13u

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

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

**Requirement 1:**

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

* I use a for loop for which I include the pif = fork() statement as wel as all the printf() statements which print to the console whether the process running is a child or parent and what their pid is.
* I use the variable NUMBER\_OF\_PROCESSES which is globally defined as 5 to control how many times the for loop executes, and hence how many times the fork() call is executed.

**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)

* Included the <sys/wait.h> header and used the function waitpid(pid, &status, WUNTRACED) before the end of the code so that parent waited for all the child processes
* WCONTINUED reports status of any continued child process specified by pid whose status has not been reported since it continued from a job control stop.

**Requirement 3:**

General comments:

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

* Before the for loop which calls fork(), I initiated the start time
* The function ‘getDifferenceInMilliSeconds’ calculates the difference between start time and current time
* The function ‘ChildProcess’ places the index of the child process next to the corresponding run time (position in array) in the array, it takes the current time and start time passed through, where the start time is always the same as the start time initialised
* For every child created, a while loop continues to get the current time and call ChildProcess until the time difference reaches the MAX\_EXPERIMENTAL\_DURATION, until this condition is false, ChildProcess is called to place values into the array
* I made use of shared memory between the processes to ensure the array would not be overwritten, this ensured synchronisation
* Initialised a pointer to the long int array, which was created to be shared using the mmap function

**Requirement 4:**

General comments:

* Used CPU\_SET to select which core to run on
* Made it to pick random core between the 8 by putting the core number as rand() % 7
* Which picks a random number between 0 and 7
* I then used sched\_setaffinity which uses the process pid to set that process to run on the chosen core

**Requirement 5:**

General comments:

* Used a switch statement to check the index of the process
* assign it its corresponding priority using the setpriority function

**Requirement 6:**

General comments:

* Increased the amount of processes to be created
* The child priority running ps -1 would have index 5 (the last child)
* In the switch statement which checks indexes and marks priority based on this index, for case with index 5 I wrote the statement ‘system(“ps –l > PROCESSLIST1.txt”) which runs the command ps -1 and outputs to PROCESSLIST1.txt

**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

* Opened a text file and wrote the SVG header to it
* Then looped through each place in the pRunTime array (holding all the run times and process indexes), printing to the file the line   
  “fprintf(file, "<rect x='%d' y='%d' width='3' height='%d' style='fill:pink'/>\n", i\*constantWdith, (pRunTimes[i] - 1) \* constantHeight, constantHeight);
* x denotes the time in milliseconds (so time is going forward as you move across the screen)
* y denotes the process number, so the first line would be the first process, second line would be the second process etc
* the pink shows the process running, whilst white shows it is not
* The array was already shared between the processes and so there was no interference

**Requirement 8:**

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

* The process scheduler is pre-emptive
* I recognise the round robin algorithm as each process has a time slice, and when a process has used up it’s time slice, the following process continues and all processes run before coming back round to the first process. The length of the time slice seems to be dependent on the priority, as when the process priorities are not equal, the process with priority 0 has larger time slices than process’ with other priorities (and so runs for longer before the CPU switches to the next process). When priorities are equal, the scheduler is reduced to FIFO as the time slices are equal and run in the order they were added in the queue.
* Starvation does not and cannot occur as in the round robin algorithm, processes take it in turn to run as if the queue were a circle.