Concurrent Programming CW2 Submission Overview

In this coursework I attempted and completed stages 1 and 2. I tried to make headway with the disk option of stage 3 but was making very little progress for the amount of time I was spending on it so discontinued that section.

Stage 1a

This stage was relatively simple given that the lab work covered most of the work that needed to be done here.

Stage 1b

This stage required more effort. Initially I implemented a round-robin that iterated over my PCB table to accomodate a flexible max number of initial processes for the generalised implementation of stage 1a.

For a priority scheduling algorithm to improve this, I chose to use a multi-level feedback queue. This comprised of 3 linked lists of differing priorities, with the nodes in the lists associated with processes in the PCB table that are ready to be executed. The process priority can be edited but is defaulted to 1, so all processes are initially placed in the highest priority queue. Lower priority queues allow programs more time to execute before switching. If a program does not finish executing before its allocated time slot it is moved to a lower priority queue. The top two queues are FCFS queues and the bottom operates as a round robin.

In theory, this scheduling system allows for very quick processes like I/O to be initialised and get executed very quickly, staying in the highest priority queue, whilst slower processes are given longer time slots at a lower priority. In the case of this coursework the processes are fairly slow and so the scheduler quickly evolves into a round-robin scheduler if no new processes are added.

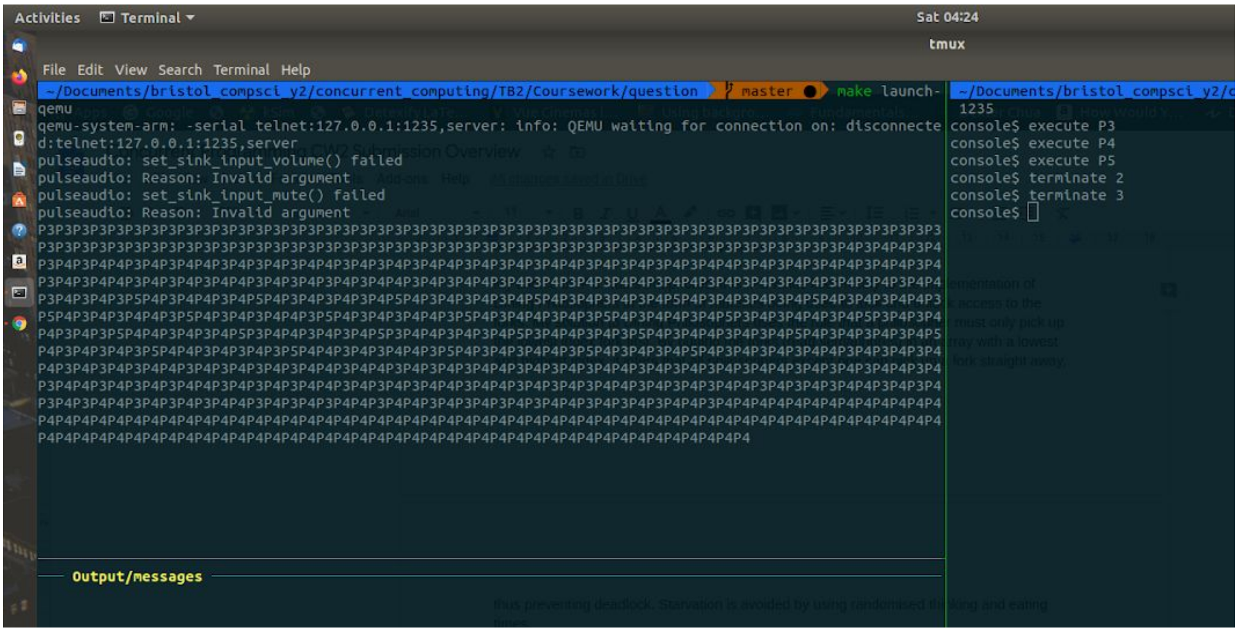
Stage 2

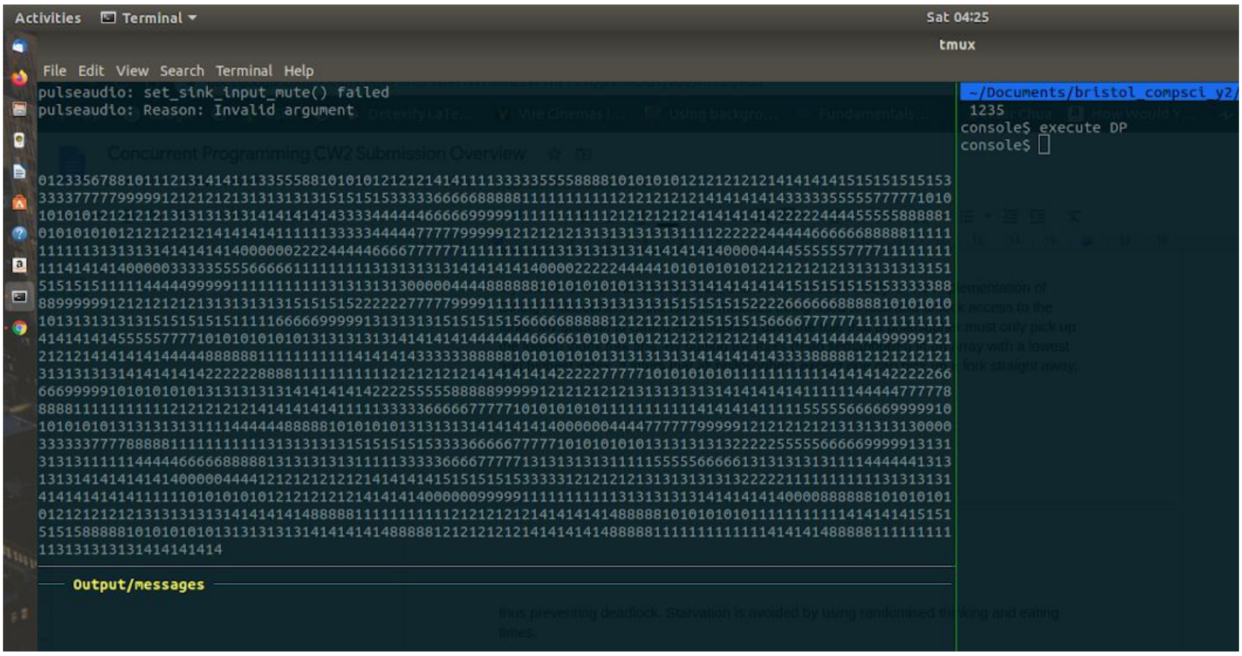
This stage took me a very long time. Dealing with the technicalities of forking processes and assigning values to the correct context and register spaces was a large hurdle for me to overcome. However, once I understood the use of memory I was able to finish this section, with the Dining Philosophers problem not too difficult to implement once I had the forking and executing system calls working properly.

My choice of IPC was semaphores; this IPC lends itself nicely to the implementation of Dining Philosophers in the form of mutexes being used to lock and unlock access to the forks. My solution to Dining Philosophers uses the rule that a philosopher must only pick up the lowest index fork first. By putting the forks (read semaphores) in an array with a lowest and highest index, it infers that all philosophers except one can pick up a fork straight away, thus preventing deadlock. Starvation is avoided by using randomised thinking and eating times.

In the first image below you can observe the use of the execute and terminate console instructions to dynamically start and end the execution of processes. Note that P5 stops without the need of a prompt because it finishes execution on its own.

In the second image you can observe the Dining philosophers problem working, with all 16 sub-processes getting initialised and getting their turn to “eat” - this is signified by the printing of the philosopher’s index.





Stage 3

Although I didn’t make much material progress in this section, attempting to create a file structure was a fun project and I will probably continue to explore it outside of the coursework.