SYSC4001 Assignment 1

Part I - Concepts

Jason Van Kerkhoven

100974276

Brydon Gibson

100xxxxxx

October 27th, 2016

Van Kerkhoven (100974276)
Gibson (100123456)

Part A
There are four different ways that the CPU will stop running the current process, process *(a)*, and start running the new process, process *(b)*.

Case 1 occurs when process *(a)* completes within 1 quantum of time, where a quantum is the time-slice that each process is allowed to use the CPU for. Process *(a)*, being complete, then voluntarily hands control of the CPU back to the dispatcher, where it will then hand control of the CPU to the next waiting process (process *(b)*).

Case 2 occurs when process *(a)* runs for its allocated quantum of CPU time. It will then be interrupted by the dispatcher, and move back into the Round Robin queue. Its inserted location is dependent on its priority relative to all the other processes waiting for CPU time. For instance, if the incomplete process *(a)* has a lower priority than all other processes in the Round Robin queue, it will be placed at the back of the queue. Otherwise (assuming process *(a)* has priority *p*), process *a* will be placed such that all processes ahead of it have priority *n>p*, whereas all processes behind it will have priority *n<p*.

Case 3 occurs when an interrupt is called, in which the incomplete process *(a)* is put onto the front of the queue, waiting for the interrupt process to terminate, where it will resume running.

Case 4 occurs when process *(a)* requires something from I/O. Process *(a)* will then release its hold on the CPU, where the dispatcher will give the next waiting process, process *(b)*, use of the CPU. When the I/O operation is complete, the CPU is given back to process *(a)*, where it will continuing running for a maximum of 1 quantum.

Part B
User threads are supported above the kernel level, and are managed without support from the kernel. Kernel threads, however, are supported and managed by the operating system directly. For instance, there can be multiple user threads running on top of a kernel thread. The advantage of running directly on a kernel thread (ie one user thread is run on-top of each kernel thread) is it allows for the maximum amount of process concurrency. Additionally, each thread is able to run in parallel on systems with multiple processors. By running on a user thread (ie having the possibility of more than one user thread per kernel thread), you risk having all user threads running becoming unable to run if one of them makes a system call (say for any I/O). They are also all unable to run truly in parallel, as only one will be executed by the kernel thread at a time.

Part C

i)
$$t_{av} = \frac{\sum_{i=1}^{n} t_i}{n}$$
$$t_{av} = \frac{t_1 + t_2 + t_3 + t_4 + t_5}{5}$$
$$t_{av} = \frac{0 + (22 - 9) + (33 - 12) + (45 - 13) + (56 - 17)}{5}$$
$$t_{av} = 21\ seconds$$

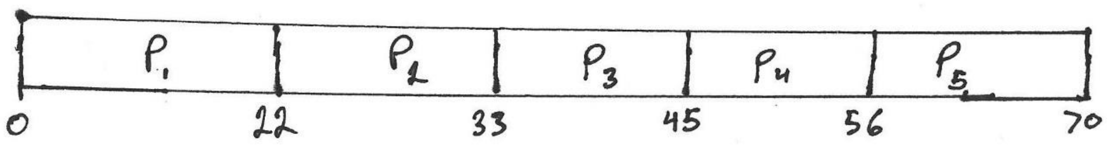

Figure 1 - Gantt chard using First-Come First-Serve algorithm

ii) $t_{av} = \frac{\sum_{i=1}^{n} t_i}{n}$

$$t_{av} = \frac{\begin{array}{c}0 + (15-12) + (24-15) + (39-27) + (53-42) + (64-56) + 0 + (18-12) + (33-21) + (48-36) \\ +(15-12) + (30-18) + (45-33) + (59-48) + (21-13) + (36-24) + (50-39) + (62-53) \\ +(27-17) + (42-30) + (58-45) + (65-59)\end{array}}{22}$$

$$t_{av} = \frac{192}{22}$$

$$t_{av} = 8.\overline{72}\ seconds$$

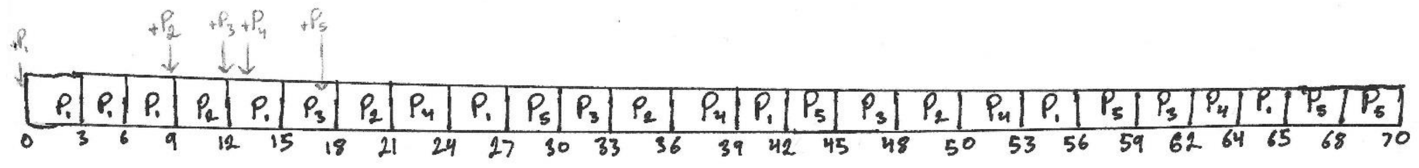

Figure 2 - Gantt chart using Round Robin algorithm with a quantum of 3 seconds

iii) $t_{av} = \frac{\sum_{i=1}^{n} t_i}{n}$

$$t_{av} = \frac{\begin{array}{c} 0 + (23 - 11) + (43 - 27) + (63 - 47) + (11 - 9) + (14 - 12) + (27 - 16) + (47 - 31) + (63 - 47) \\ +0 + (16 - 13) + (31 - 18) + (51 - 35) + (66 - 55) + 0 + (19 - 14) + (35 - 21) + (33 - 39) \\ +(18 - 15) + (21 - 19) + (39 - 23) + (59 - 43) + (67 - 63) \end{array}}{23}$$

$$t_{av} = \frac{188}{23}$$

$$t_{av} = 8.1739$$

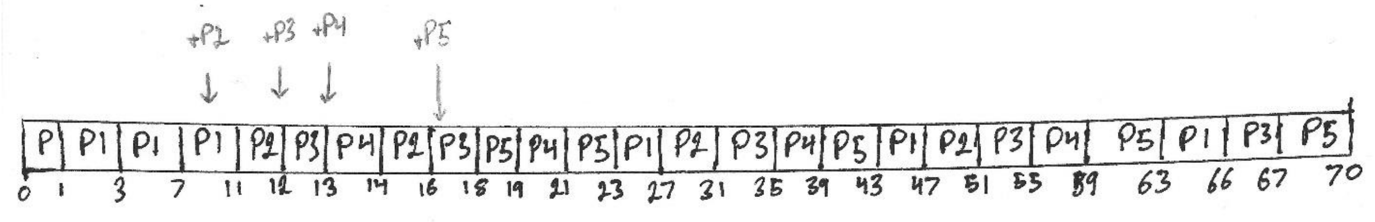

Figure 3 - Gantt chart using multiple queues with feedback. Primary queue quantum at 1s, secondary queue quantum at 2s, and tertiary queue quantum at 4s.

Part D

Part E

a) The First-Come First-Serve (FCFS) scheduling algorithm do not discriminate for time consuming processes, that is, process length greatly affects the average turnaround time. As soon as a long process begins execution by the CPU, it will not stop until complete or until I/O is called. Due to this, no other process will be given time on the CPU, even if said processes only need a few seconds of CPU time to compute. The algorithm makes no distinction between a lengthy process, and a short one.

b) The Round Robin (RR) algorithm somewhat discriminates against time consuming processes. Since it cuts off each process after 1 *quantum* of time, no matter the process length, a single time-consuming process cannot monopolize the CPU. However, if the time *quantum* is too large, then the RR algorithm effectively will become a FCFS algorithm (due to the long time consuming processes never being interrupted, as they never exceed the *quantum*). Therefore, longer processes take more time to fully execute, as the CPU is frequently switching to execute shorter $[quantum \cdot (n - 1)]$, where n is the number of processes.

c) Multilevel Feedback queues discriminate heavily against time consuming processes, as any process that takes longer than the specified quantum for the primary queue is shifted down to the secondary queue. The secondary queue will only begin execution once the primary queue is empty. If the process still is not complete after the quantum for the secondary queue has been reached (the quantum of the secondary queue being greater than that of the primary queue), the process will be placed in the tertiary queue, which similarly only executes when both the primary and secondary queue are empty. From this, it can clearly be seen how this algorithm heavily discriminates against any time consuming process.

Part F