CS3500: Operating Systems

Lab 6

Date: 6th Sep 2019

In this lab, we will use the setup you have been building in the previous lab to simulate a couple of common scheduling algorithms.

1. In the previous lab, you worked with a single workload process. As the first step, modify your code to work with multiple workload processes. Here are the changes to be made:

   (a) G would now be given a list of (a, m, n, p, t) tuples and for each tuple it should generate a workload process. Implement this by reading the job descriptions from an input file, where each line contains five numbers corresponding to a, m, n, p, and t for each job. Notice that we have added a which refers to the arrival time of the job, in addition to m, n, p, and t as defined earlier.

   (b) The function of W would be the same, with one minor change. As soon as W starts, it goes becomes inactive for a duration of a. This simulates a late arrival. Only after W wakes up, does it signal S about its creation for the first time.

   (c) Each W would still communicate with S as before.

   (d) For simplicity, you can assume that G communicates with S after creating all W and updating the shared memory to provide details of number of processes and their parameters to S. On its side, S waits for this communication before proceeding with anything else.

   (e) S now gets signals from multiple Ws. To decide which process was the originating process, use the $sigaction()$ system call to map signals to handler.

   (f) The state transition for a W from Ready to Running is now not immediate. Indeed, it happens only when that W gets to execute as decided by S.

   (g) S must decide which process to run at specific times. Let us call these times, *scheduling instances*. To start with, these include times when a 'running' W either completes or goes into an emulated IO call. At these times S calls its $schedule()$ function which decides which W to run next.

2. After finishing the above setup, implement the First-Come-First-Serve policy. Test it out with a few configurations of jobs. In particular, try out a combination of jobs that can demonstrate the *convoy effect* we saw in the lecture, i.e., where a long CPU-intensive job stalls all remaining jobs increasing the average turn-around time for all.

3. To visualise the schedule, it helps to plot it out. Use the `plot.tex` file to do this. This file assumes as input a file `schedule.csv` which contains three numbers per line, separated by commas. The first number denotes the start time of the execution of a job, the second the end time of that CPU allocation (not necessarily the completion time), and the third number denotes the job number (starting from 1 to N). Modify the $schedule()$ function in S to output such a

`schedule.csv` file. Also modify `plot.tex` with two parameters in lines 5 and 6 corresponding to the number of jobs and the total time of plotting. Hint: Output times in a unit that is easy to visualise. Now plot the schedule for the chosen setup that demonstrates the convoy effect.

4. After FCFS, its now the turn of the Round Robin scheduler. This requires a parameter, the time-slice. Set the time slice to 100 ms. Also note that the scheduling instances must be now changed to also include the time instance when the time-slice expires. Implement this with the $settimer()$ system call in S. For the example you used for the convoy effect, draw out the corresponding schedule with Round Robin scheduler. Does the average turn-around time decrease?

5. Design the job set to have two CPU intensive tasks and two IO intensive tasks. Execute the job-set with both FCFS and RR schedulers. Plot the schedules and comment on average turn-around and response times.

6. Think how you would implement a preemptive fixed priority policy. What additional events must be added to the scheduling instances? Time permitting, implement the fixed priority policy.