CS330
Assignment - 2
Team Name:

amogOS

Soham Samaddar
200990

Aditya Tanwar
200057

Samarth Arora
200849

Sarthak Kohli
200886

September 2022

1 Preliminary Setup

An `extern` variable defined in `param.h` is used to store the current scheduling policy since this variable has to be accessed across many files. There is an outer infinite loop which ensures that the scheduling process occurs continuously. The outer infinite loop uses a switch statement to choose the correct scheduling algorithm and begins executing the same.

2 Non-Preemptive Shortest Job First Scheduler

First, we make the algorithm non-preemptive by disabling the yield function call in the timer interrupts. This is done by simply checking if the current scheduling policy is SJF. If it is, then do not call `yield()`. We create a variable to store the next process to be scheduled and a variable to store the minimum estimated burst length encountered thus far. In the scheduler, we run across all the processes in the process table. Before we do any calculations, we first check if the current scheduling algorithm has changed. If it has changed, break out of the inner loop to the outer infinite loop so that the new scheduling algorithm may be applied. Now we check find a process which is `RUNNABLE`. If the process is not from the batch (determined using the `from_forkp` variable in the process table), we immediately schedule the process. Note that this scheduling is done inside the inner loop so that after we context switch back to the inner loop again, we continue to scan the remaining processes in the processes table (and not start from the 0^{th} process again). Otherwise, if the process is a batch process, then we check the estimated CPU burst length. If it is the smallest among all the estimated CPU burst lengths till now (or if it is the first estimated CPU burst length), then store this process as the next process to be scheduled. After a run through is done across the entire process table, we have the process with the minimum estimated CPU burst length. If there were no such processes, we do not schedule anything and continue to the next iteration of the infinite loop. Otherwise, we schedule the process.

3 Preemptive UNIX Scheduler

First, we run through the entire process table. If we find a `RUNNABLE` process from the batch, then we update the `cpu_usage` and `priority` of the process (both stored in the process table) as given in the assignment. After this first run is done, we create a variable to store the next process to schedule as well as the minimum priority value encountered thus far. We loop through the process table again. When we find a `RUNNABLE` process, we check if it is from the batch. If not, schedule it inside the loop. Otherwise, we check the priority of the process. If it is the smallest among all the priorities till now (or if it is the first priority), then we store this process as the next process to be scheduled. After a run through the entire table is done, we have the process with the minimum priority value (and hence the maximum priority). If there were no such processes, we do not schedule anything and continue to the next iteration of the infinite loop. Otherwise we schedule the process. Note that we iterate through the process table twice in this algorithm. For *both* the inner loops, we always check at the beginning if the scheduling algorithm has changed.

4 Batch Statistics

		Batch 1	Batch 2	Batch 7
Batch Execution Time		9018 *vs.* 9065	8984 *vs.* 9121	9100 *vs.* 9049
Average Turn-around Time		9015 *vs.* 9049	8981 *vs.* 9103	5000 *vs.* 9040
Average Waiting Time		8108 *vs.* 8137	8082 *vs.* 8191	4090 *vs.* 8134
Completion Time	(Avg)	9014 *vs.* 9048	8979 *vs.* 9102	4999 *vs.* 9039
	(Max)	9016 *vs.* 9063	8981 *vs.* 9117	9098 *vs.* 9048
	(Min)	9012 *vs.* 9021	8979 *vs.* 9017	909 *vs.* 9024

Table 1: FCFS *vs.* RR

For `batch1.txt` and `batch2.txt`, there is not much difference between `FCFS` and `RR` since the processes frequently give up their CPUs willingly, either by `sleep()` or `yield()`. Both `FCFS` and `RR` act as fair schedulers.

In `batch7.txt`, `RR` continues to schedule processes fairly through the timer interrupt. However, since `FCFS` is non-preemptive and does not yield the CPU on a timer interrupt, it finishes each process *one by one*. All the processes get scheduled initially, but each process completes entirely before allowing the next process to begin execution. Hence there is a big variance between the *minimum and maximum completion times*.

		Batch 2	Batch 3
Batch Execution Time		9075	36464
Average Turn-around Time		6579	25236
Average Waiting Time		5672	21590
Completion Time	(Avg)	6578	25234
	(Max)	9073	36461
	(Min)	3440	9514
CPU Burst	(Count)	55	207
	(Avg)	164	176
	(Max)	191	200
	(Min)	1	1
CPU Burst Estimates	(Count)	63	213
	(Avg)	161	174
	(Max)	191	200
	(Min)	1	1
CPU Burst Estimates Error	(Count)	45	197
	(Avg)	21	9
Ratio		0.1280	0.0511

Table 2: SJF

We notice that the processes submitted in `batch3.txt` contain approximately 4 times more CPU bursts than `batch2.txt`. This explains why the absolute statistics (the *average* values and the *count* values) also follow the same ratio. The order statistics (*minimum* and *maximum*) are more skewed.

The *average* value of the estimation error is much smaller for `batch3.txt` than `batch2.txt`. This is due to the fact that each process in `batch3.txt` is able to generate a greater number of estimates (owing to its longer execution time) which allows the estimate to approach the actual value. In fact, both the *absolute error* has decreased from 21 to 9 and the *fractional error* (the ratio) has decreased from 0.1280 to 0.0511.

		Batch 4
Batch Execution Time		6824 *vs.* 6865
Average Turn-around Time		6821 *vs.* 4546
Average Waiting Time		6138 *vs.* 3860
Completion Time	(Avg)	6819 *vs.* 3936
	(Max)	6822 *vs.* 6863
	(Min)	6818 *vs.* 908
CPU Burst	(Count)	- *vs.* 54
	(Avg)	- *vs.* 127
	(Max)	- *vs.* 190
	(Min)	- *vs.* 1
CPU Burst Estimates	(Count)	- *vs.* 60
	(Avg)	- *vs.* 129
	(Max)	- *vs.* 190
	(Min)	- *vs.* 1
CPU Burst Estimates Error	(Count)	- *vs.* 44
	(Avg)	- *vs.* 13

Table 3: FCFS *vs.* SJF

`SJF` has a much smaller *average turn-around time (ATT)* and much smaller *average waiting time (AWT)* compared to `FCFS` which is as expected since `SJF` provably attains the minimum ATT and AWT. However, the difference between the *minimum and maximum completion time* is stark in `SJF` while they are essentially the same in `FCFS`. This alludes to the fact that `SJF` is not a fair scheduler, while `FCFS` is comparatively more fair. In `SJF`, the scheduler picks the `testloop3.c` processes frequently to complete its execution as soon as possible since these processes are shorter compared to `testloop2.c`.

		Batch 5	Batch 6
Batch Execution Time		9067 *vs.* 9088	9010 *vs.* 9068
Average Turn-around Time		9046 *vs.* 5978	8999 *vs.* 5966
Average Waiting Time		8135 *vs.* 5065	8098 *vs.* 5059
	(Avg)	9045 *vs.* 3928	8999 *vs.* 5351
Completion Time	(Max)	9066 *vs.* 6143	9008 *vs.* 9065
	(Min)	8971 *vs.* 2730	8961 *vs.* 2729

Table 4: RR *vs.* UNIX

Between `RR` and `UNIX`, `UNIX` has a smaller *average turn-around time (ATT)* and *average waiting time (AWT)* while `RR` has a small variance between the *minimum and maximum completion times*. This is due to the fact that `RR` is the most fair scheduler, while `UNIX` tries to find a middle ground between fairness and performance.

Between `batch5.txt` and `batch6.txt`, `RR` does not show much difference. However, there is a sharp increase in the *maximum completion time* and *average completion time* in `UNIX`. This occurs since in `batch5.txt`, the process calls `sleep()` while in `batch6.txt`, the process calls `yield()`. In `batch6.txt`, the processes with *low base priority value* will be continuously scheduled while the processes with *high base priority value* will not be scheduled frequently. This situation does not arise in `batch5.txt` since the higher priority processes go to sleep, forcing the CPU to process the lower priority processes as well.