SAN DIEGO STATE UNIVERSITY

Computer Engineering 571
Professor Aksanli
Programming Assignment #2

Ethan Nagelvoort, RedID: 821234668
Ivan Cruz, RedID: 823794173

How Process Response Time Was Calculated

To calculate the response time for a process, we took the difference of when the algorithm starts and the time the process finishes. We used timeval structs to capture the time the algorithm starts, capture the time the process ended, and to store the difference of the start time and end time. For example, in each algorithm, you will find that we initialize the start time by implementing the *gettimeofday()* function. We also use this function to get the timing of when the processes end and store it in an array of timeval structs, end[4]. Then timersub() is used to get the difference between start time and end time and store the result in struct res. Then, for each process, we calculate response time with $(res.rev\_sec * 1000000 + res.tv\_usec)/1000000$. End time - start time will be equivalent to execution time + wait time in our algorithms and so will accurately give response time. Each algorithm's description will go into some more detail of how timing is done but this is generally how it was accomplished.

How Each Algorithm Was Implemented

The sample code provided the RR algorithm. Average runtime was calculated by having a timeval struct array called end[4]. Each index of this array is used to capture the time at the end of each if statement. These if statements are within a while loop that will continue to loop if there is at least one process still running. This means that each index of end[4] will capture the time when each process ends before the program exits the while loop. Start time was captured when the algorithm began. After the while loop, a for loop was used to calculate each process's runtime. Then average runtime was calculated and printed out.

To implement SJF, we first looked at the workloads and determined which process had the least amount of work to do. Then, to change the sample code, we rearranged the kill(pid#, SIGCONT) commands so that the process with the least work went first and the process with the most work went last. Then waitpid(pid#, &running#, 0) is called to block the program until the current process finishes running. Once the process finishes, the running# will turn to 0. This command is within a while loop based on running# >0 to check that running# is 1. Usleep() functions were removed since quantums were not used. Kill(pid#, SIGSTOP) commands were removed since we did not need to stop processes. The if statements were also removed. The start struct and end[4] structs capture start and end times of the processes. After, the runtime for the processes is calculated in a for loop. At the end, the average of the runtimes is calculated and printed out.

To implement FCFS, we first looked at the order of which process was created first, second, third, and last. Then we rearranged the kill(pid#, SIGCONT) commands in the sample code to reflect that order. waitpid(pid#, &running#, 0) was also used to block code. Similar to SJF, the usleep() functions, kill(pid#, SIGSTOP) commands, and the if statements were removed. The procedure to calculate the runtime of each process and the average runtime overall was also similar to the SJF procedure.

To implement MLFQ, we must have queue 1 operate with RR and queue 2 operate with FCFS. We added to the RR given sample code by removing the while loop and moving each process' waitpid() after the if statement that corresponds to it. Then after each waitpid(), there's

another if statement that checks if that process is still running. If it is not, then the end time of the process is saved in struct end[4]. This is so that if the process finishes in queue 1, we can capture endtime immediately. Also note that start time was captured at the beginning of the algorithm. Then for queue 2, we simply copied and pasted the FCFS code we had already coded. We added if statements that checked if the process was running in queue 2 since they could have finished in queue. Average runtime was calculated at the end and printed out.

Data Structures, Assumptions, and How code was ran

We used arrays to hold the end times of the processes in end[4] and to hold the total run times when calculating average runtime, t[4] . One assumption made while coding all the algorithms was that we assumed that all the processes arrived at the same time (ie. arrival time = 0). This meant that calculating a process's run time involved capturing the time at the start of the algorithm. Also, by looking at the sample code, we could assume that pid1, pid2, pid3, and pid4 were created in that order. We also assumed that the workload corresponded to the execution time of each process. Hence the bigger the workload the more execution time. This helped us in our coding for SJF. Each of the experiments below involved running the algorithms but to get accurate data, we had to make sure the CPU was not being used up. We closed all applications besides Ubuntu and we observed the task manager application which allows us to view what percent of the CPU was being used up. We ran the code only when this percentage was low.

Experiment 1 (Round Robin (RR)):

RR	Q	AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG[1-5] (sec)	STD of Averages
	10	4.936317	4.830274	4.955494	4.841148	4.824706	4.8775878	0.06301078
	100	1.945281	1.949381	1.942269	1.931477	1.955894	1.9448604	0.00905792
	1000	1.525774	1.528828	1.558564	1.546803	1.596912	1.5513762	0.02876622
	10000	1.531989	1.513036	1.488125	1.517978	1.500126	1.5102508	0.01682319
	100000	1.644621	1.642896	1.63749	1.648358	1.653369	1.6453468	0.00595159
	10000000	3.769141	3.76219	3.764611	3.761046	3.765436	3.7644848	0.00315009
b/t 1k & 10k	5000	1.541657	1.491145	1.510969	1.502089	1.496232	1.5084184	0.01998906
b/t 5k & 10k	7500	1.483125	1.510293	1.493802	1.519978	1.516875	1.5048146	0.01578853
b/t 5k & 7.5k	6200 (rounded)	1.508545	1.506843	1.491125	1.499621	1.483661	1.4979590	0.01054618
b/t 6.2k & 7.5k	6800 (rounded)	1.493322	1.483854	1.498935	1.514271	1.492421	1.4965606	0.01127363
b/t 6.2k & 6.8k	6.5k	1.495716	1.483678	1.504973	1.516456	1.744491	1.5490628	0.10990945
	QUANTUM = 6.8k							

Figure 1) Data for AVG & STD with respect to (Q) Quantum required to obtain optimal Q Value.

In this experiment, we found that Quantum=6.8k gave us the optimal time result which is 1.496 secs. To obtain these results, we tested a variety of quantums and ran the code five times with them, resulting in getting five different response time averages. Thus, the code was run 55 times. Then the overall average of those averages was calculated. We first increased the time quantum by a factor of 10 to observe which two gave the smallest average response times and from there we tested the midpoints of those quantums. This was then repeated until the overall average response time converged to a small value. Different quantum values have processes divide the workload differently, resulting in different average response times. If the quantum is too high, then RR prioritizes longer workloads making processes with smaller workloads wait longer. This resembles FCFS. If the quantum is too low, then there is more context switching

overhead and larger workloads wait longer for smaller ones to finish. Both of these will result in higher average response times. Giving different quantums to each process will give certain processes more priority than others to finish. Hence processes with smaller quantums will wait more for processes with high quantums. For example, higher quantums could be used for higher workloads so processes with smaller workloads would wait longer, resulting in higher average response times.

Experiment 2 (Multilevel Feedback Queue (MLFQ)):

MLFQ	Q	AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG[1-5] (sec)	STD of Averages
	10	3.283153	3.218958	3.226995	3.1199618	3.231387	3.21609096	0.0593701
	100	3.19732	3.226191	3.255394	3.378357	3.242342	3.25992080	0.0696656
	1000	3.283816	3.198666	3.283766	3.274618	3.270129	3.26219900	0.0360069
	10000	3.352591	3.239863	3.319553	3.212653	3.234498	3.27183160	0.0606575
	100000	2.646749	2.58467	2.591795	2.668962	2.608950	2.62022520	0.0363262
	10000000	3.658627	3.667429	3.656933	3.658422	3.658772	3.66003660	0.0041974
b/t 10 & 100k	50k	2.483067	2.458638	2.495633	2.486987	2.475080	2.47988100	0.0139918
b/t 50k & 100k	75k	2.544583	2.530248	2.563927	2.550717	2.570003	2.5518956	0.0157806
b/t 50k & 75k	62k	2.493975	2.527949	2.517845	2.544478	2.50731	2.5183114	0.0193073
b/t 50k & 62k	56k	2.478882	2.495656	2.500692	2.507579	2.541379	2.5048376	0.0230107
b/t 50k & 56k	53k	2.46855	2.485557	2.510178	2.501064	2.502584	2.4935866	0.0166091
b/t 50k & 53k	51k	2.471341	2.468165	2.515272	2.467376	3.33293	2.6510168	0.3817311
	QUANTUM = 50k							

Figure 2) AVG & STD with respect to (Q) Quantum required to obtain optimal Q Value.

In this experiment, we found that our best Average Response time of 2.479 second***s*** happens at Quantum Value = 50k**.** The process of finding the best quantum value was similar to the last experiment where we tested midpoints of a variety of quantum values. Hence, the code was run for a total of 60 times. If the quantum is too low then the FCFS (queue 2) part of MLFQ does most of the work and if it is too high, then the RR part (queue 1) of MLFQ does most of the work. It would be better to have both queue 1 and queue 2 do the work evenly so tasks with smaller workloads have an opportunity to finish sooner. If RR did most of the work, then the quantum would be high and so this RR algorithm could be dangerously close to resembling a FCFS. Hence, with the current workloads, smaller workload processes would have to wait longer for higher workload processes to finish. This is also true if FCFS did most of the work. Therefore, both of these setups would not be good. It is not good to give different quantum sizes in the first level queue because it is important to have low workload tasks finished in queue 1 to have a lower average response time. Different quantums could make low level tasks enter queue 2 resulting in them having more wait time. Also if the higher workload processes have a higher quantum, then smaller workload processes would have too much wait time in queue 1.

Experiment 3a: (Comparison of all Algorithms)

CASE 3A								
RR	Quan	AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG[1-5] (sec)	STD of Averages
	6.8k	1.498284	1.482189	1.620699	1.496869	1.489446	1.5174974	0.05804916
MLFQ	Quan	AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG[1-5] (sec)	STD of Averages
	50k	2.4654565	2.47735825	2.48785075	2.4818435	2.4906485	2.4806315	0.00993632
FCFS		AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG [1-5] (sec)	STD of Averages
		3.221687	3.198876	3.224976	3.240817	3.238668	3.22500480	0.01681095
SJF		AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG [1-5] (sec)	STD of Averages
		1.174576	1.175707	1.197744	1.187101	1.188054	1.1846364	0.00962431

Figure 3) Data pertaining to Response Time Averages and STD amongst all Algorithm

We found that SJF gave the smallest response time average when testing all of the algorithms. Each algorithm was run five times. SJF giving the smallest response time supports the theoretical information seen in class since in this algorithm, smaller processes are guaranteed to finish first before longer processes. This results in the least amount of wait time.

Experiment 3b: (Comparison of all Algorithms with workloads equal to 100000)

CASE 3B	WORKLOAD[1-4] = 100K							
RR	Quan	AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG[1-5] (sec)	STD of Averages
	6.8k	10.392146	10.452803	10.423754	10.455203	10.639591	10.47269940	0.09674619
MLFQ	Quan	AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG[1-5] (sec)	STD of Averages
	50k	6.661777	6.670626	6.695561	6.588403	6.582822	6.63983780	0.05106529
FCFS		AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG [1-5] (sec)	STD of Averages
		6.711442	6.571166	6.565205	6.521507	6.619557	6.59777540	0.07242288
SJF		AVG 1 (sec)	AVG 2 (sec)	AVG 3 (sec)	AVG 4 (sec)	AVG 5 (sec)	AVG [1-5] (sec)	STD of Averages
		6.485251	6.493964	6.51766	6.498287	6.498853	6.49880300	0.01186305

Figure 4) Data pertaining to Response Time Averages and STD amongst all Algorithms

In this case, we found that the best scheduling algorithm is still SJF. This result is the same as in case 3a, however even though SJF is the best, it's average was close to the average of FCFS. This is because both of these algorithms essentially operate the same way if all workloads are the same, and so will give very similar average response times. MLFQ was similar but also longer since it involved more context switching. RR involved a lot of context switching and so had high response times. Please note that each algorithm was tested 5 times. It would not be optimal, for both RR and MLFQ, to give different quantums to different processes even though each process has the same workload. The reason is similar to why you would not do it in the other experiments. Giving one process a higher quantum than the other processes will result in the other processes waiting longer for that process to finish and so will result in a higher average response time. Even though the workloads are the same in this experiment, different quantums will have processes finish off portions of the workload at different times and so some processes will take longer than others.

Standard Deviation Analysis

Standard deviation (STD) was also calculated using the different averages found per quantum. The higher the STD, the more inconsistent the average response times were. This can help measure how unreliable the data could be due to error. There are higher STD values for smaller quantums because the time required for context switching could differ and so lead to inconsistent response times. These high STD values come from more context switching occuring. It takes time to save the context of one process and then access the context of another. This timing can vary for each context switch. Hence, there is notably much higher STD values within the RR algorithm due to the fact that more context switching happens compared to other algorithms. RR, thus, has more of a chance for error to occur. SJF and FCFS have shorter STD values due to less contact switching and MLFQ has more due to more context switching. When comparing experiments 3a and 3b, we can also see that higher workloads lead to a higher value of STD for each algorithm. This is because there is more of a chance for background processes to cause CPU usage to increase, interfering with the algorithm's response time.