Report for Part 1 Interrupt simulator: Skanda Nagendra 101299202, Aaron Fisher 101294988

The implemented interrupt simulator tracked the following 4 metrics: Throughput, Average Turnaround Time, Average Wait Time, and Average Response Time. This was done by adding variables that tracked different aspects of each scheduling algorithm during runtime. A map had to be included so that the PID of each process was linked to their respective turnaround/wait/response times. Suppose we want to track response times. We need to know each process's first run time (not tracked) and its arrival (which is already part of the PCB). As soon as a process enters the running state (state transitions from READY->RUNNING) for the first time, the current time is recorded, then linked to the PID, and pushed to a vector of all response times recorded for a particular test file. After all process's are complete, the program uses this vector of response times to calculate the average. Likewise, a similar approach of tracking specific state transitions is applied to calculate the wait and turnaround times.

Results of all 4 metrics are then printed out on the console after running each test file, which have been recorded into the following table:

	THROUGHPUT (Processes/sec)			AVERAGE TURNAROUND TIME (ms)			AVERAGE WAIT TIME (ms)			AVERAGE RESPONSE TIME (ms)		
TEST FILE	RR	EP_RR	EP	RR	EP_RR	EP	RR	EP_RR	EP	RR	EP_RR	EP
test_cpu_bound1	4.98753	4.98753	4.98753	350	350	300	150	150	100	50	50	100
test_cpu_bound2	4.98753	4.98753	4.98753	350	300	300	150	100	100	50	100	100
test_cpu_bound3	6.65189	6.65189	6.65189	350	350	250	200	200	100	50	50	100
test_cpu_bound4	4.99168	4.99168	4.99168	466.667	466.667	466.667	266.667	266.667	266.667	100	100	266.667
test_cpu_bound5	19.9005	19.9005	19.9005	125	125	125	75	75	75	75	75	75
test_io_bound1	4.07332	4.07332	4.07332	480	480	480	10	10	10	10	10	10
test_io_bound2	4.80769	4.80769	4.80769	382.5	382.5	382.5	42.5	42.5	42.5	0	0	0
test_io_bound3	2.52845	2.52845	2.52845	787.5	787.5	787.5	2.5	2.5	2.5	2.5	2.5	2.5
test_io_bound4	5.39568	5.39568	5.39568	350	350	350	100	100	100	81.6667	81.6667	81.6667
test_io_bound5	6.65189	6.65189	6.65189	420	420	420	70	70	70	30	30	30
test_mixed_case1	7.96813	7.96813	7.96813	200	200	237.5	60	60	97.5	12.5	12.5	12.5
test_mixed_case2	3.99202	3.62976	3.62976	380	375	375	105	100	100	15	100	100
test_mixed_case3	3.99202	3.99202	3.80228	380	380	377.5	105	105	102.5	15	15	15
test_mixed_case4	6.23701	6.10998	6.10998	346.667	335	335	143.333	131.667	131.667	28.3333	98.3333	98.3333
test_mixed_case5	4.53515	4.38596	4.38596	305	302.5	302.5	77.5	75	75	25	75	75
test_mixed_case6	2.49377	2.49377	2.49377	400	400	400	0	0	0	0	0	0
test_mixed_case7	8.54701	8.31025	8.31025	183.333	170	170	46.6667	33.3333	33.3333	0	16.6667	16.6667
test_mixed_case8	2.3245	2.60281	2.60281	1070	1136	1136	322	388	388	68	106	106
test_mixed_case9	2.3245	2.35738	2.35738	1070	1040	1040	322	292	292	68	0	0
test_mixed_case1	2.37982	2.34632	2.34632	1048	1048	1048	300	300	300	188	174	174

Note*: For EP and EP_RR to work properly, we had to modify the PCB to include a priority member which EP and EP_RR checks for in the input files. RR ignores this value.

CPU Bound Processes:

- Throughput:
  - Test 1-5: All three processes have similar throughput compared to each other. Generally, they all increase as the number of processes being tested increases. Tests 3 and 4 are noteworthy since T4 has a slightly lower throughput, likely due to all the processes arriving at the same time, while T3 has them arriving at different times. Test 5 however has the highest throughput since each process has a small enough CPU burst that fits within RR's time slice, and equal priority, so EP and EP_RR have no sorting to do beforehand, so all 3 algorithms can quickly complete and switch between processes.
- Average Turnaround Time (TT/TAT)
  - Test 1: RR and EP_RR have equal TAT values. This is because the priority of both processes are equal, so the only element pre-empting each process is RR's time slice. This midway pre-emption results in a higher TAT compared to the EP algorithm, since that method only worries about the priority of each task, and allows the whole process to complete.
  - Test 2: EP and EP_RR have equal TAT values since P2 has a higher priority than P1 and both arrive at the same time, meaning P2 will be completed faster than P1, resulting in a generally lower TAT. RR does not take priorities into account, so its TAT is identical to Test 1
  - Test 3: EP has lower TAT value than RR and EP_RR, since it only worries about the priority of each process, and does not pre-empt any of them midway. Even though the arrival times are different, TAT for RR and EP_RR are identical to Test 1 since all the processes have the same CPU burst times. The staggered arrival time actually favours EP because P2 and P3 arrive later and must remain in Ready state for less time – compared to tests 1 and 2, where all processes show up at the same time so all process in READY have to wait longer.

    - Test 4: TAT for all three process is the same, because A: all processes have the same priority, so EP simply completes all the processes one by one, and B: although RR pre-empts all the processes every 100ms, since no other factors involved like I/O, the total time each process takes in the CPU is the same as other algorithms. EP_RR follow the same logic as the independent EP and RR algorithms, so it also has the same TAT as the other two. Generally, they all have a higher TAT than test 3, because the total workload is higher (T4: 100+200+300 = 600ms of CPU, T3: 150 x 3 = 450ms)
    - Test 5: The TAT values are the same between each algorithm, because of the similar reason to Test 4. Only now, each process is much smaller, so they are completed much more quickly, resulting in lower TAT than Test 4.
- Average Wait Time (WT)
    - Test 1: RR and EP_RR have the same wait times since both processes are being pre-empted evenly every 100ms, causing the total wait time to build up. EP has only to worry about the priority, and since both processes are equal, they terminate one after another, without waiting as long. Having equal priorities between processes the EP algorithm in this case.
    - Test 2: EP and EP_RR have the same wait time since both P1 and P2 have the same priority and arrive at the same time. P2 finishes faster than P1. In the EP_RR algorithm, although RR would pre-empt P2 after 100ms, it still has the higher priority, so EP would override RR's decision and reallocate it to the CPU, which is the same as letting it run uninterrupted like the EP algorithm. RR would take longer because of repeated pre-emption, building up the wait time compared to EP and EP_RR
    - Test 3: RR and EP_RR have the same WT since the dominant element of these algorithms is the RR pre-empting and building up the wait times for each process (a similar logic to Test 1). Again, EP here only worries about the priorities, and since they are equal, the total wait time is relatively lower than RR and EP_RR
    - Test 4: The same logic explaining the equal TAT times applies here: EP completes all processes one by one, and the lack of I/O between all processes means they all spend the same time waiting for the CPU regardless of the algorithm chosen
    - Test 5: Similar reasoning to Test 4.
- Average Response Time (RT)
    - Test 1: The RT for RR and EP_RR are the same here since both processes have equal priority, so the priority element has no real effect in deciding which process to choose next. Since the time slice for RR is 100, one process is allocated at T = 0, and the other at 100, so both processes enter RUNNING state faster than EP, while EP has to let one of the process wait for the full 200ms, thus increasing EP's response time.
    - Test 2: RR's response time is faster than EP and EP_RR due to the same reason as Test 1. In EP_RR, since P2 has a higher priority here, EP's decision to prioritize P2 override's RR's 100ms pre-emption, making P1 wait just as long as it does in the independent EP algorithm.
    - Test 3: The staggered arrival times + equality of the CPU bursts of the 3 processes work in favour of the RR and EP_RR algorithms. Since all of them have equal priority, RR only makes each process wait for 50ms, resulting in an avg of 50ms until the first response. EP on the other hand just makes all of them process until completion, making its RT higher
    - Test 4: All processes arrive at the same time, but with varying lengths, so RR is able to pre-empt every 100s, making the processes only wait for 100 to 200ms, making RR's and EP_RR's RT only 100ms. EP processes them until completion, making P2 and P3 wait 300ms and 500ms respectively. Processes that arrive at the same time work in favour of RR
    - Test 5: RT of all processes are the same. All the processes arrive at the same time + have equal CPU bursts.

I/O Bound Processes:

- Values for throughput, TT/WT/RT are all identical between all three algorithms. This is an interesting result, but it's explainable. All the I/O bound tests have an I/O frequency that is less than RR's time quantum value, so RR never gets a chance to actively pre-empt any of the processes. In the case of EP, any variation in priorities between processes does not make a significant difference. Yes, EP still determines the initial order of which ones to process, but due to the frequent I/O tasks, each process gets allocated to the CPU little by little, thereby evenly processing each task until completion. Example: test_io_bound5 prioritizes P1>P3>P2. Due to the frequent I/O, EP continues to process them one after the other, operating almost like an FCFS algorithm. Due to this lack of pre-emption of either algorithm, all metrics between EP, EP_RR and RR are identical.

Mixed Cases:

- Throughput: Thorughput values are nearly identical for almost all cases, with RR having a slight advantage in some of the time due to it actively pre-empting tasks, and giving fair chances to other processes. EP and EP_RR do have a slight advantage when there are several (4+) processes involved + staggered arrival times + varying and frequent I/O.

- Average Turnaround Time (TT/TAT)
    - Test 1: EP has a disadvantage since it cannot actively pre-empt tasks
    - Test 2: because P1 is prioritized, EP and EP_RR can focus on completing P1 before P2, which decreases their TAT compared to RR alone. RR's fair allocation is mainly handy when multiple processes vary in CPU bursts
    - Test 3: EP has a slight advantage since it focusses on completing the process that has I/O, while RR and EP_RR ends up splitting allocation more evenly, increasing the overall TAT
    - Test 4, 5: Similar logic to Test 2 - EP and EP_RR can focus on completing higher priority tasks, thus reducing TAT
    - Test 6: only one process – all TAT are same
    - Test 7: Staggered arrival times work in favour of EP, since the difference between arrival and their completion times are less than the case of RR
    - Test 8, 9, 10: TAT is in favour of EP and EP_RR as their TAT does not increase as drastically as the complexity and number of the processes increase.
- Average Wait Time:
    - Test 1: Because EP cannot pre-empt tasks, other process are left waiting for longer. This is a problem if the running task has a high CPU burst. EP is at a disadvantage
    - Tests 2, 3, 5, 6, 10: differences in wait times are minor, almost negligible, or they are identical
    - Test 4, 7: Staggered arrival times + priority work in favour of EP and EP_RR
    - Test 8, 9 : RR's performance remains unchanged since the only real difference is the priorities being reversed. Reversing the priority surprisingly improves the wait times for EP/EP_RR. Since Test 8 gives higher priority to early processes, EP/EP_RR makes later process wait longer, increasing its wait time drastically. However, when the priorities are reversed, later processes immediately get access to CPU once the earlier processes need to do I/O. This is likely what's making the wait times spread more evenly between each process.
- Average Response Time
    - Test 1, 3, 6: RT values are identical
    - Test 2: RR's response time is drastically better. Since both process arrive at the same time, RR chooses P2. P2 only runs in CPU for 30ms until I/O is needed, so P1 gets allocated immediately. EP/EP_RR has to give P1 priority, so P2 has to wait at least 100ms before being allocated
    - Test 4: the staggered arrival times work in favour of RR, since P2 and P3 arrive later, they wait for less time before being allocated to CPU
    - Test 5: EP/EP_RR are at disadvantage since P2 has to wait till P1 is finished (150ms -> 150/2 = 75ms). RR pre-empts P1 after 100ms, so P2 only waits for 50ms (50/2 = 25ms)
    - Test 7, 8: RR is in favour
    - Test 9: Like the average wait time metric, EP/EP_RR perform better when later processes have higher priority. It is also the case that here, earlier processes have very frequent I/O, while later ones have longer I/O frequencies. This gives the later processes more frequent windows to enter the Running, letting them be serviced as soon as they arrive. RR does not have any say with prioritizing tasks, nor does it ever get a chance to pre-empt since all processes have an I/O frequency value that's lower than its time quantum.
    - Test 10: EP/EP_RR have slight advantage, but it is not relatively significant