CS330 Assignment 2

Shubhan R	Manas Gupta	Shivam Malhotra	Dishay Mehta
200971	200554	190808	200341

October 26, 2022

1 Implementation of SCHED_NPREEMPT_SJF

There is a variable, *schedalgo* in **proc.c**, which stores current algorithm. To disable timer interrupts from usertrap and kerneltrap functions of **trap.c**, I have modified the yield() function in **proc.c** to have an integer argument. Yield is called from 3 places. By user program yield() call, and by usertrap and kerneltrap functions of **trap.c**. I have each of these 3 passes a unique integer to yield() in **proc.c**. Depending on the *schedalgo* and the argument passed to yield(), it may return directly without executing. This is how I have disabled timer interrupts in FCFS and SJF algorithms, by disabling calls from usertrap and kerneltrap of **trap.c**.
I have added 3 new variables, last_cpu_burst, last_scheduled, previous_estimate to **proc.h**. All these are set to 0 initially. last_scheduled is set to current ticks every time it is scheduled. In **proc.c**, I have added a new function, *calc_estimate_burst*, which calculates the next predicted cpu burst of the current process according the the formula given. The last_cpu_burst is updated in the sys_yield(), sys_sleep() of **sysproc.c**, and exit() of **proc.c**, as (current ticks - last_scheduled). *previous_estimate* is also updated to the next estimate in the same functions using *calc_estimate_burst*. The scheduler then goes over all the batch processes and chooses the one with least estimated burst (*previous_estimate* variable). However, if it encounters a non batch process, it is scheduled immediately.

2 Implementation of SCHED_PREEMPT_UNIX

I have added 3 new variables, *base_priority*, *cpu_usage* and *priority* to **proc.h**. Everytime yield() or sleep() is called by the program, *cpu_usage* is updated according to the way explained in the question. Everytime the scheduler runs, it goes over all RUNNABLE processes, halves *cpu_usage* and sets *priority* to *base_priority*+*cpu_usage*/2. Then it goes over all processes to see if there is any non batch process and schedules it immediately. If there is no such process, it iterates over all the batch processes and schedules the process with least priority. This is how I have implemented UNIX algorithm.

3 Evaluation

Comparison between non-preemptive FCFS and preemptive round-robin:

- Evaluate batch1.txt for SCHED_NPREEMPT_FCFS and SCHED_PREEMPT_RR. Report the differences in statistics. Explain the observation.
  - **FCFS**
    Batch execution time: 14053
    Average turn-around time: 13094
    Average waiting time: 14325
    Completion time: avg: 14046, max: 14054, min: 14039
  - **RR**
    Batch execution time: 13602
    Average turn-around time: 12850
    Average waiting time: 11594
    Completion time: avg: 13449, max: 13603, min: 13292

- Evaluate batch2.txt for SCHED_NPREEMPT_FCFS and SCHED_PREEMPT_RR. Report the differences in statistics. Explain the observation.
  - **FCFS**
    Batch execution time: 13687
    Average turn-around time: 12964
    Average waiting time: 11674
    Completion time: avg: 13683, max: 13688, min: 13678
  - **RR**
    Batch execution time: 13347
    Average turn-around time: 12342
    Average waiting time: 11097
    Completion time: avg: 13187, max: 13348, min: 13032

- Evaluate batch7.txt for SCHED_NPREEMPT_FCFS and SCHED_PREEMPT_RR. Report the differences in statistics. Explain the observation.
  - **FCFS**
    Batch execution time: 11542
    Average turn-around time: 5992
    Average waiting time: 4910
    Completion time: avg: 6656, max: 11543, min: 1759
  - **RR**
    Batch execution time: 12071
    Average turn-around time: 11379
    Average waiting time: 10243
    Completion time: avg: 12041, max: 12072, min: 12004

batch1 has testloop1. We can visualize each process(if it ran alone) as a blocks of 5 CPU bursts with short I/O intervals. We also see that the time quantum of RR is smaller than one of the CPU bursts. We know that FCFS has an unbounded waiting time and RR optimizes for waiting time. This is visible here also in the results. Both complete in range of average completion time as both algorithms go over all runnable processes cyclically one by one (either due to cooperation or due to timer interrupt) and all end one after the other in a small time range. Further explanation is given in **batch7** explanation of this same question.

batch2 has similar observations and explanations as given for **batch1** above. Here, each process is considered as a contiguous block of 5 CPU bursts.

batch7 contains 10 testloop4 processes and we know that testloop4.c program has no I/O burst at all except one at the end. Each process can be visualized as a block of one large CPU burst followed by a very small IO burst. As there is only single CPU burst, FCFS completes the whole process, then moves on to the next process. So, there is huge difference between completion time of the first process and last process compared to average, unlike in the case of batch1 and batch2 where each process was cyclically scheduled over a long time frame(compared to one of the cpu bursts). But RR has very small range in completion time because it cyclically schedules processes for a time quanta and the time quanta is much lesser than the total time it takes to complete the batch. The interesting observation is the contrast in turnaround time and waiting time between FCFS and RR. We will use the model of process as described before of one long cpu burst. Suppose length of each CPU burst is x and length of time quantum is y. Assume x is a multiple of y for simplicity. The batch completion time B is 10x. The average turnaround time derived for FCFS is: 5.5x. We get this by (x+2x+3x+...10x)/10. For RR, it is B - 4.5y which is 10x - 4.5y. Last process completes at B, the second last completes at B-y and so on. Thus We get (10x+(10x-y)+(10x-2y)...(10x-9y))/10 = 10x - 4.5y. Now, if(y/x<<1) as in the case here, we see that turnaround time for FCFS is 5.5x, but for RR, it is comparably greater than 5.5x. This is what we observe in results also. Similarly, we can derive waiting times also and explain the difference in a similar way. For batch1, and batch2

also the times can be derived mathematically and explained similarly. In this case, the ratio of y/x is closer to 1, and putting in the derivations, we get similar times for both FCFS and RR.

CPU burst estimation error using exponential averaging:

- Evaluate batch2.txt and batch3.txt for SCHED_NPREEMPT_SJF and report any observed differences in the CPU burst estimation error. You should be paying attention to how the following ratio changes: (the average CPU burst estimation error per estimation instance)/(the average CPU burst length). Explain what you observe.

  – **Batch2**
  Batch execution time: 13426
  Average turn-around time: 11017
  Average waiting time: 10638
  Completion time: avg: 12850, max: 13427, min: 11178
  CPU bursts: count: 59, avg: 211, max: 321, min: 1
  CPU burst estimates: count: 59, avg: 188, max: 257, min: 108
  CPU burst estimation error: count: 49, avg: 100

  – **Batch3**
  Batch execution time: 43999
  Average turn-around time: 32915
  Average waiting time: 29454
  Completion time: avg: 52295, max: 61866, min: 42195
  CPU bursts: count: 208, avg: 208, max: 235, min: 1
  CPU burst estimates: count: 208, avg: 202, max: 226, min: 107
  CPU burst estimation error: count: 198, avg: 21

  First, we see from the code that a CPU burst length of both testlooplong **(batch3)** and testloop2 **(batch2)** are approximately same. This is reflected in average CPU burst length of results also. We observe that the batch execution time for the `Batch3` is larger as compared to `Batch2`, which is obvious given that the `Batch3` contains `testlooplong` processes which consists of larger number of CPU bursts interspersed with yield() calls while `Batch2` consists of *testloop*2 processes which have shorter number of CPU bursts with yield() calls.
  We see that testlooplong has 4 times more CPU bursts than testloop2 which is reflected in the batch execution time also. As testlooplong process has longer runtime, this is reflected in the average turnaround time also. As the average CPU burst length is same for both, let us compare average CPU burst error per estimation directly. t(n) which is actual CPU burst length is the same value, lets say x for all n. The initial estimate s(1) is 0. s(1) is x/2. s(2) is 3x/4. The more iterations happen, the closer and closer s(n) gets to x which is the real value. As testlooplong has 20 iterations of the CPU burst as compared to 5 iterations in testloop2, the estimations get better and better. This, and the fact that the denominator count while dividing is larger for testlooplong, it gives lesser estimation error than testloop2.

Comparison between non-preemptive FCFS and non-preemptive SJF:

- Evaluate batch4.txt for SCHED_NPREEMPT_FCFS and SCHED_NPREEMPT_SJF. Report the differences in statistics. Explain the observation.

  – **FCFS**
  Batch execution time: 9338
  Average turn-around time: 8293
  Average waiting time: 7470
  Completion time: avg: 9335, max: 9339, min: 9330

  – **SJF**
  Batch execution time: 8910

Average turn-around time: 5242
Average waiting time: 5306
Completion time: avg: 75319, max: 77721, min: 73225
CPU bursts: count: 60, avg: 139, max: 245, min: 1
CPU burst estimates: count: 60, avg: 126, max: 221, min: 54
CPU burst estimation error: count: 50, avg: 63

We observe that batch4.txt contains 5 testloop2 and 5 testloop3 processes alternatively with same priority and we know that testloop2.c program has long CPU burst with yield() and testloop3.c is similar to testloop2.c but with shorter CPU burst.
Let the lengths of the long and short bursts by x and y respectively. We also see that x$\sim$2y in the code. In FCFS, x and y alternate till completion of the batch. In SJF, x and y are scheduled alternatively till one cycle of all processes is over as all the estimated lengths are currently 0. The calculated burst lengths of testloop2 is x/2 and for testloop3 is y/2 now. After that, the algorithm only schedules testloop3 cyclically as it always has smaller predicted bursts (never greater than y, which is approximately x/2) than testloop2 till all the testloop3 completes. Then it schedules all testloop2 cyclically till completion. Due to this, we see that half of the processes with small CPU bursts complete earlier than half of the processes with longer CPU bursts. Therefore, the average turnaround time of SJF is smaller than FCFS. This also means that in SJF the wait times of testloop3 are much smaller than that of tesloop2. This causes SJF to have smaller average wait time than FCFS where they are more or less evenly distributed among testloop2 and testloop3. The large difference in max and min completion time in SJF also can be explained similarly, as the first testloop3 to complete, completes significantly earlier than the last testloop2 to complete as compared to FCFS.

Comparison between preemptive round-robin and preemptive UNIX:

- Evaluate batch5.txt for SCHED_PREEMPT_RR and SCHED_PREEMPT_UNIX. Report the differences in statistics. Explain the observation.

  - **RR**
    Batch execution time: 11907
    Average turn-around time: 11229
    Average waiting time: 10142
    Completion time: avg: 11893, max: 11908, min: 11879

  - **UNIX**
    Batch execution time: 12430
    Average turn-around time: 4088
    Average waiting time: 6163
    Completion time: avg: 29551, max: 33293, min: 25637

- Evaluate batch6.txt for SCHED_PREEMPT_RR and SCHED_PREEMPT_UNIX. Report the differences in statistics. Explain the observation.

  - **RR**
    Batch execution time: 11428
    Average turn-around time: 10885
    Average waiting time: 9797
    Completion time: avg: 25237, max: 25256, min: 25200

  - **UNIX**
    Batch execution time: 12005
    Average turn-around time: 4081
    Average waiting time: 6133

Completion time: avg: 46618, max: 50363, min: 42713

First thing we observe is that the results are consistent for both algorithms across both **batch5** and **batch6**. This is because each instance of testloop1 can be visualized as 5 blocks of CPU bursts followed by IO bursts. The IO interval is smaller than CPU burst here. So, IO bursts can be aligned well with CPU bursts here. Each instance of testloop2 can be visualized as 5 contiguous blocks of CPU bursts. So, from analysis point of view, both are identical cases as IO bursts can align well with CPU bursts. So, explaining any one batch would be fine. We will explain **batch5**. In UNIX algorithm, we see that the priorities given to each process is different. So, the least base priority process will be scheduled first, followed by next smallest base priority, ...so on the processes with least net priorities will be scheduled. The CPU usage decays at the same rate for each process over time. So, the main differentiator here is the base priority. Few processes complete early as their base priority is very less. Once they are over, the middle processes have a chance at completing, and after this, the high base priority processes have a chance of completing. So, in an ideal case, each process can not complete unless the all the processes with lesser base priority have completed. So, the turnaround times (and therefore completion times as they are scheduled at same time) are similar to if they were scheduled one after the other continuously without yielding in between. This explains why UNIX has a large range in completion times. As some processes complete way earlier than others, the average turnaround time is lesser than RR, where all of them complete almost at the same time. Due to differences in base priorities, processes with smaller base priorities have lesser waiting time compared to larger base priority as they have a good chance of being scheduled earlier. This also causes the lesser average waiting time for UNIX than RR.