Scheduling Project

Introduction:
This project focusses on implementing a real time scheduler which schedules a set of tasks that have specific compute times and are periodic. This implementation includes rate monotonic scheduler, earliest deadline first and least slack time schedulers. A resulting analysis is also provided which analyzes the performance of each scheduler.

Description:
For this project, we employed a scheduler thread, a timer thread, an idle thread, a thread each for all tasks in addition to the main thread. Scheduler thread is run at a high priority which is 10. Timer thread signals scheduler thread every 5 milliseconds to schedule a task for the next period, using pthread conditional wait and signal routines. Based on the type of scheduler in use, the scheduler determines the task with highest priority at that particular scheduling point. For rate monotonic scheduler, the scheduler determines the task with the shortest period and if it is not finished for its period, that particular task thread is assigned highest priority and remaining task threads are assigned lower priority, so the task with highest priority runs for the next cycle. For earliest deadline scheduler, at each scheduling point the next deadline relative to the current execution cycle is determined for each task and the task with earliest deadline is assigned highest priority and remaining task threads are assigned lower priority. Since this is a dynamic scheduler, this process is followed and task thread priorities are changed at every scheduling point, i.e whenever the timer thread signals the scheduler thread to schedule. Similarly for least slack time scheduler, at each scheduling point slack time is calculated for all tasks and the task with least slack time is assigned highest priority and remaining task threads are assigned lower priority. Since this is also a dynamic scheduler priorities of task threads are changed dynamically.
Each task is defined with a compute time and deadline. Since these tasks are periodic, period and deadline of each task are the same. These task threads are implemented in such a way that they consume CPU cycles for their respective compute time. This is done using nanospin. This is used because it simulates a real thread that is running that doesn't have to have any special modifications to the way it runs. The scheduler thread also checks for any missed deadline at each scheduling point, as well as the start of a new period for any thread. Execution times of each task are measured by logging user events during the start and end of task execution. Also user events are logged whenever deadline is missed. Therefore resulting analysis is done by capturing the kernel event log trace. The type of scheduler can be changed in the program by just changing a macro, which can be set to rate monotonic scheduler, earliest deadline scheduler or least slack time scheduler. Threads can also easily added using the thread_meta_t struct. This struct holds all the information the user is assigning to a thread such as execution time, period, and deadline.

The program is run on 5 task data sets, of which three sets are schedulable, one is failure task and one is a set with identical tasks. Execution time, average execution time and number of deadlines missed for each task set and each scheduler is listed out in the Results spread sheet. Following are the task data sets used.
Task set 1: (1,7,7), (2,5,5), (1,8,8), (1,10,10), (2,16,16)
Task set 2: (1,3,3), (2,5,5), (1,10,10)

Scheduling Project

Task set 3: (1,4,4), (2,5,5), (1,8,8), (1,10,10)
Failure Task set: (3,4,4), (2,6,6), (3,9,9)
Identical Task set: (3,10,10), (3,10,10), (3,10,10)

Task set 1 has a CPU utilization of 0.89. For rate monotonic scheduler, the total number of deadline misses in 500 cycles is 4. And for earliest deadline first and least slack time schedulers, it is 0 and 6 respectively. Earliest deadline first scheduler works best for this task data set. The jitter on task execution times range from 0.5 to 0.9 ms over a period of 5 ms for all three schedulers.
Task set 2 has a CPU utilization of 0.83. Rate monotonic and earliest deadline schedulers never missed deadline for tasks in 500 cycles. Least slack time scheduler failed thrice in meeting the deadline of tasks. Both rate monotonic and earliest deadline schedulers perform well in this case. The jitter on task execution times range from 0.6 to 1 ms over a period of 5 ms for all three schedulers.
Task set 3 has a CPU utilization of 0.65. The rate monotonic scheduler missed the deadline 7 times and earliest deadline scheduler misses twice. But the least slack time scheduler has zero misses in 500 cycles, by which we can say that least slack time scheduler performs best among the three schedulers for this case. The jitter on task execution times range from 0.5 to 1 ms over a period of 5 ms for all three schedulers.
For the failure task data set, tasks missed the deadline for 110, 116, 107 times respectively for rate monotonic, earliest deadline and least slack time schedulers. Since the tasks are not schedulable, scheduler successfully fails for this case. Also we can note that earliest deadline has the best failure rate among the three schedulers.
For the identical task data set, tasks never missed the deadline even with 90% CPU utilization. And earliest deadline scheduler performed the best for this case.
For rate monotonic scheduler, the scheduler overhead ranges from 25 to 40 micro seconds. And for earliest deadline scheduler and least slack time scheduler, it ranges from 30 to 50 micro seconds. In the current program scheduler overhead of rate monotonic scheduler can be improved, as currently it calculates the priority at every scheduling point which is not necessary as it is not a dynamic scheduler. We can also improve the earliest deadline first and least slack time schedulers by optimizing the dynamic algorithms.
From the above results, it can said that theory and experimental results do not match as in real time systems, scheduler overhead needs to be taken into consideration. It is important to make sure that the scheduler logic has least possible memory access instructions as it would add to scheduler overhead. Choosing one scheduler over another entirely depends upon the task data set.

Conclusion:
This project helps in analyzing the performance of all three schedulers on various task data sets and in getting experience in implementing a scheduler in real time applications. It can be inferred from the results that theoretical and experimental results do not entirely match because of scheduler overhead which can improved by developing efficient dynamic algorithms.