Matthew Thomas/Kelsey Hecker
CSC 623
HW3 - Scheduler Simulation

Program Description

The design of our scheduler simulation program is intended to be as straightforward as possible. Our program first processes the inputs and parses the input task file, allowing us to assemble a list of periodic and aperiodic tasks that will need to be simulated. We then implemented three major pieces of functionality. The first piece was our static priority scheduling algorithms for periodic tasks. The second piece was our dynamic priority scheduling algorithms for periodic tasks. The last piece was our aperiodic task scheduling. No matter if we are using a static or dynamic priority scheduling algorithm, we always schedule the periodic tasks first because we know that they will always have higher priorities than any aperiodic tasks. Once all the periodic tasks are scheduled we can then begin to schedule the aperiodic tasks in the remaining slack time as they arrive.

Static Priority Scheduling of Periodic Tasks

There are three static priority scheduling algorithms in use in our program. They are the First Come First Serve algorithm, the Rate Monotonic algorithm, and the Deadline Monotonic algorithm. The way static priority algorithm tasks are scheduled in our program is by placing them into a presorted list of all the jobs across the entire simulation time window based on their priorities.

This list is generated by determining the priority of the original task list based on the selected scheduling algorithm. Then for each task all the jobs that will fit within the entire simulation time window are generated and appended to a list. This results in a list of all the jobs that is already sorted by priority and order of arrival.

Then this list of jobs is iterated over and added one by one to the simulation schedule. Each subsequent job in the list is of a lower priority than the previous job meaning that once a job has been inserted into the schedule it will not be changed as it is guaranteed to be higher priority than anything else that could be inserted into the list. If a lower priority job is being inserted into the schedule but does not have enough room to fit in its entirety then the excess portion of the job will be sliced off and inserted in the next possible opening. This process is repeated until attempts have been made to insert every job in the list.

Dynamic Scheduling of Periodic Tasks

There are two dynamic priority scheduling algorithms in use in our program. They are the Earliest Deadline First algorithm, and the Least Slack Time algorithm. For the dynamic priority algorithms there is no need to presort the list based on priorities, as they will be dynamically

calculated every time a new task becomes available, a task completes execution, or after one second elapses in the simulation. That being said the list of jobs will be generated using the same process as for the static priority algorithms, but for the dynamic priority algorithms all the jobs are sorted solely on their arrival times.

Our dynamic scheduling process follows three major steps. The first step is to grab all the jobs in our full job list and add them to our available job pool. This step also determines the next point in time a job will be available. The second step is to calculate the priorities of all the jobs in the available job pool based on our current dynamic scheduling algorithm and set the highest priority jobs to be our currently executing job. The third step is to attempt to execute that currently executing job until either the job finishes execution, a new job becomes available, or 1 second has passed, whichever happens sooner will stop execution and return us to the first step. These steps are repeated until the current time exceeds the entire simulation time.

It is of note that if our current time is not a whole number and the current job is able to run the full 1 second, we will execute the full 1 second instead of running to the next whole number.

Scheduling of Aperiodic Tasks

Once all of the periodic tasks have been scheduled we can begin to schedule the lower priority aperiodic tasks. The aperiodic tasks have a similar insertion implementation to the static priority periodic tasks, but have additional limitations. These limitations come in the form of the server execution budget and replenishment schedule.

The program first determines how many server periods are needed to cover the entire simulation time window and assigns each server period a start time, an end time, an execution budget, and the first task start time in the given server period. Giving each server period its own execution budget imitates the replenishment process that would occur at the beginning of each period.

Once the server periods are established the program attempts to insert the aperiodic task into the schedule. The scheduler will iterate over the server periods and attempt to find a server period that the aperiodic task could feasible land in, as well as has execution budget remaining. The polling server introduces an added constraint with the new task having to be within a certain time window of the first task to be scheduled in the given server period. If the scheduler determines that the given server period has available execution budget to execute the aperiodic task, then the scheduler will search for open space in the simulation schedule between any existing periodic tasks. If there is no room in the current period or if there is remaining execution time for the aperiodic task, the whole process is repeated again in the next server period until either the aperiodic task is fully executed or the simulation ends.

Summary

In summary, our program reads in the input parameters and then parses the input tasks and places them into either the periodic task pool or the aperiodic task pool. Next we sort the periodic task pool based on the scheduling algorithm we have selected. For the static priority scheduling algorithms we then insert the static priority sorted tasks into the simulation schedule one after the other knowing that each subsequent task is of a lower priority than all the existing tasks in the schedule. For the dynamic priority scheduling algorithms we simulate the tasks as they arrive and dynamically calculate their priority when

Test Case Analysis

All test cases were run with a server period of 20 seconds and server execution budget of 5 seconds.

Test Case 1

Test case 1 has a system utilization of 1.1 and a system density of 1.236. Both those values being greater than one indicates that we will have a difficult time finding a feasible schedule for the task set. In fact the high utilization and density numbers seem to act as good indicators of whether or not we will be able to schedule our aperiodic tasks, which we were not able to do using any of the periodic scheduling algorithms.

The FCFS algorithm missed 27 of the 36 potential deadlines. The first few tasks executed fine, but very quickly it became apparent that the tasks with short periods, Task 1 and 2, were already missing their deadlines. Task 3 and 4 had longer periods and were able to more frequently beat their deadlines. Overall the mix of tasks with relatively short periods and relatively long periods was not a good mix for the FCFS algorithm.

The Rate Monotonic algorithm only missed 3 of the 36 potential deadlines. All the deadline misses were for Task 4, which would suggest that without Task 4 we would be able to have a feasible schedule using the RM algorithm.

The Deadline Monotonic algorithm has identical results to the Rate Monotonic algorithm because the deadlines are almost identical to the periods causing the results to be the same.

The EDF algorithm missed 19 of the 36 potential deadlines. The results indicate that earlier in the schedule tasks were able to narrowly meet their deadlines, but the more deadline misses that accumulated the more likely it became that there would be more deadline misses. The high density and utilization of this test case can be seen as an indicator of this problem.

The LST algorithm missed 23 of 36 potential deadlines. The results here closely match the results when using the EDF algorithm and we can draw a similar conclusion.

Overall, this test case shows that the FCFS algorithm is ill suited to schedule tasks with varying period lengths. This test case shows that the EDF and LST algorithms struggle with task sets that have high density and utilization and that once deadline misses begin to accumulate they become more and more likely to occur. This test case suits the RM and DM algorithms well and with lower densities and utilizations they could very likely schedule all their tasks. Unfortunately, this test case shows that a density and utilization greater than one will prevent the scheduling of aperiodic tasks.

Test Case 2
Test Case 3
Test Case 4

Summary