SAN DIEGO STATE UNIVERSITY

Computer Engineering 571
Professor Aksanli
Programming Assignment #3

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

RM Algorithm and EE RM Algorithm

In this algorithm, we use a struct to hold information about the first row of the input file and an array of structs to hold info on the individual tasks. This info includes execution time, WCET1, WCET2, WCET3, WCET4, status, etc. We get a lot of these values by parsing the input file. First we check if we are doing the energy efficient RM algorithm and if so, the program goes through five different for loops to cycle through each task's WCET1, WCET2, WCET3, and WCET4. We take each of these execution times and divide them by the tasks deadline. Inside the last for loop, we add all of these values up into tTot and compare it to double ff. This double is equivalent to the value 5(2^(⅕)-1). This equation is what grants a feasible RM schedule. tTot has to be lower than ff or it is not energy efficient and will miss deadlines. tTot is also compared to old_tTot which is an old version of tTot from a previous loop that was successful. If this tTop is bigger than old_tTot, then we make tTot equal to old_tTot. tTop is equivalent to the utilization of the schedule and so if tTot is bigger then there will be less idle time in the algorithm, granting less energy consumed. Once a suitable tTot is found, then for each task, we set its execution time to the appropriate execution time used to find the correct tTot. We also set the correct power and frequency(i.e. 1188, 918, 645, 384) for each task struct. If the tTot comparisons always fail, then the program outputs that there is no energy efficient RM schedule for the input file. If we are not doing EE RM, then we set each task's execution equivalent to its WCET1, its frequency to 1188, and its power to its power at 1188 Hz. We then also compare the tasks utilization to the RM inequality and determine if there is a sufficient RM schedule. After, we use a bubble sort algorithm to sort the tasks in the struct array in accordance to its priority. The bubble sort algorithm sorts according to each task's deadline. The smaller the deadline, the more priority the task has. Before the while loop is described, we will describe all the different functions. RM_remainder() is used to determine the correct execution if the task currently being executed is going to be interrupted by a task with higher priority. RM_deadline() is used to update the deadlines of each task if time passed its deadline. This also changes a status variable in the task structs to 1 if time passed the deadline. RM_exec() updates the execution time for each task by examining status which will have been changed in RM_deadline(). Status changes to 0 in this function. nextTask() is used to determine the next task to be executed by examining the executions of each task in order of priority and setting a flag variable equivalent to that task. If no task has execution left, then the flag is set to idle. This flag variable will be what the switch case statement described shortly will be based on. Back to the while loop, there is a switch case statement that has each case represent each task in order of priority. The last case is an idle case. Also the while loop ends when time is greater than the CPU execution time (1000). Each case, except the idle case, generally operates the same way. First, we use the RM_remainder() within an else if statement to determine if the current task is going to be interrupted. The length of the else if statement depends on the priority of the task. The highest priority task, or case 1, does not use RM_remainder() since it will not be interrupted. The else if statement compares the current time + execution of current task to each higher priority task's current deadline to see if it will cross it. If it does not cross any deadlines, the execution is the normal value. This execution is saved in variable t. We then calculate energy of the current task using t and print out the necessary statement. The time variable is then incremented by t. After, we update all the deadlines and executions of each task. We then check if time is greater than 1000, if so then we break away from the switch case statement. Then the current tasks execution is subtracted by t. We then set t=0 and have flag equal to nextTask(). The idle case operates the same way as the other cases except with one difference. It first takes all the tasks current deadlines and puts it into an array. It then performs a bubble sorting algorithm to determine the shortest current deadline. Then the idle cases execution is equivalent to the time it takes to reach that deadline.

Results for RM Algorithm and EE RM Algorithm

```
ethan@ethan-VirtualBox:~$ ./RM.o input1.txt RM
We are using RM scheduling algorithm
The RM inequality does not work with this RM schedule
1 w4 1188 57 35.625000J
58 w2 1188 40 25.000000J
98 w5 1188 35 21.875000J
133 w3 1188 67 41.875000J
200 w4 1188 57 35.625000J
257 w2 1188 40 25.000000J
297 w3 1188 3 1.875000J
300 w5 1188 35 21.875000J
335 w3 1188 34 21.250000J
369 w1 1188 31 19.375000J
400 w4 1188 57 35.625000J
457 w2 1188 40 25.000000J
497 w1 1188 3 1.875000J
500 w3 1188 100 62.500000J
600 w4 1188 57 35.625000J
657 w5 1188 3 1.875000J
660 w2 1188 40 25.000000J
700 w5 1188 32 20.000000J
732 w3 1188 4 2.500000J
736 w1 1188 64 40.000000J
800 w4 1188 57 35.625000J
857 w1 1188 8 5.000000J
865 IDLE IDLE 15 1.260000J
880 w2 1188 40 25.000000J
920 w5 1188 35 21.875000J
955 IDLE IDLE 45 3.780000J
Total energy consumed: 591.915000J
Percentage of idle time:6.000000%
Total time of schedule: 0.000044 seconds
```

```
ethan@ethan-VirtualBox:~$ ./RM.o input2.txt RM
We are using RM scheduling algorithm
1 w5 1188 35 21.875000J
36 w2 1188 40 25.000000J
76 w4 1188 57 35.625000J
133 w3 1188 104 65.000000J
237 w1 1188 53 33.125000J
290 IDLE IDLE 10 0.840000J
300 w5 1188 35 21.875000J
335 w2 1188 40 25.000000J
375 IDLE IDLE 75 6.300000J
450 w4 1188 57 35.625000J
507 w3 1188 93 58.125000J
600 w5 1188 35 21.875000J
635 w3 1188 5 3.125000J
640 w2 1188 40 25.000000J
680 w3 1188 6 3.750000J
686 w1 1188 53 33.125000J
739 IDLE IDLE 161 13.524000J
900 w5 1188 35 21.875000J
935 w4 1188 25 15.625000J
960 w2 1188 40 25.000000J
Total energy consumed: 491.289000J
Percentage of idle time:24.600000%
Total time of schedule: 0.000038 seconds
```

Figure 1: Non-EE RM ran with input files

```
ethan@ethan-VirtualBox:~$ ./RM.o input2.txt RM EE
We are using RM scheduling algorithm
We are using the energy efficient version of this schedule
1 w5 918 45 20.115000J
46 w2 918 50 22.350000J
96 w4 1188 57 35.625000J
153 w3 1188 104 65.000000J
257 w1 1188 43 26.875000J
300 w5 918 45 20.115000J
345 w2 918 50 22.350000J
395 w1 1188 10 6.250000J
405 IDLE IDLE 45 3.780000J
450 w4 1188 57 35.625000J
507 w3 1188 93 58.125000J
600 w5 918 45 20.115000J
645 w2 918 50 22.350000J
695 w3 1188 11 6.875000J
706 w1 1188 53 33.125000J
759 IDLE IDLE 141 11.844000J
900 w5 918 45 20.115000J
945 w4 1188 15 9.375000J
960 w2 918 50 22.350000J
Total energy consumed: 462.359000J
Percentage of idle time:18.600000%
Total time of schedule: 0.000078 seconds
```

```
ethan@ethan-VirtualBox:~$ ./RM.o input1.txt RM EE
We are using RM scheduling algorithm
We are using the energy efficient version of this schedule
There was no efficient energy RM
```

Figure 2: EE RM ran with input files

Looking at our results for RM, we can see that our energy efficient RM for input2.txt is less than if run with normal RM. It also has less idle time since it has better utilization. This less idle time also grants better energy efficiency. Total execution time took longer for EE RM but that may be because execution times are for each task. When input1 was run with EE RM, there was no optimal energy efficient way found to schedule this file. This is because when the files tasks, executions, and deadlines were put through the for loops, no combination of tasks were found that could satisfy the RM inequality. When input1 was placed into the non-EE RM algorithm, it was found that the tasks execution and deadlines could not satisfy the RM inequality. This may have led to this schedule to miss deadlines when run with normal RM. It has a low idle time and that may be because of this schedule's utilization not fulfilling the RM inequality. There is less of a chance for gaps if there is no space in the CPU.

EDF Algorithm

For EDF scheduling, three different structs were made prior to beginning such that they can store information grabbed by the *fscanf* function. As mentioned in the assignment, we are given multiple rows that characterize its information. The first row of the text file includes Num_Task, Execution Time, and the active CPU power given at different frequencies. This set of data is stored in our *textTable* struct. Next, the remaining rows within the text file represent the task and its given deadline and worst-case execution time (WCET). This is stored into our *Task* struct obtaining its name, deadline/period, four WCET variables, status and remainder. Last, we created a ready queue struct *(rQueue)* that stores values of the task's arrival time and how much execution time it has left.

At this time, we'd like to first explain our main function that provides a sequence of *for loops* and *if/else statements* that provide storage for all tasks and to test which algorithm is being used. Based on a ternary operator, the code checks which algorithm will be used depending on the string provided by the user which will enable the if statement to run the EDF algorithm. When running the EDF, just like in class, we run a schedulability test based on the formula

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

If its utilization is less than one, we assume that the period is feasible therefore its period should run the task. After assumption, we can then initialize our ready list by passing in the tasks into our struct *rTask.* When each task has been initialized, we created a *nested for loop* that contains the maximum execution time of 1000 and multiple if/else statements that determines which task should run first based on deadline priority and to print its corresponding start time, name, execution time, and energy consumption on a CPU Active power of 1188MHz. To calculate Energy Consumption, we used the formula of multiplying each worst case execution time to the said 1188Mhz and dividing it by 1000(execution time). Also, we can determine its priority deadline by creating a sort function by an if statement that determines if one task's deadline is less than another. If this is true, we switch the positions between said tasks which will allow one to move to an earlier execution. As the *for loop* iterates through each task, depending on the remaining time left, our execution time counter will continue to increment while the task's remaining time decrements until it finishes. At last, we created a flag on the position of the current task. To do this, we use another if/else statement that determines if its current value is -1, it reaches an Idle state that shall produce an output of "IDLE." This will continue to loop until the variable *secs* reaches the maximum given execution time of 1000.

Results for EDF Algorithm

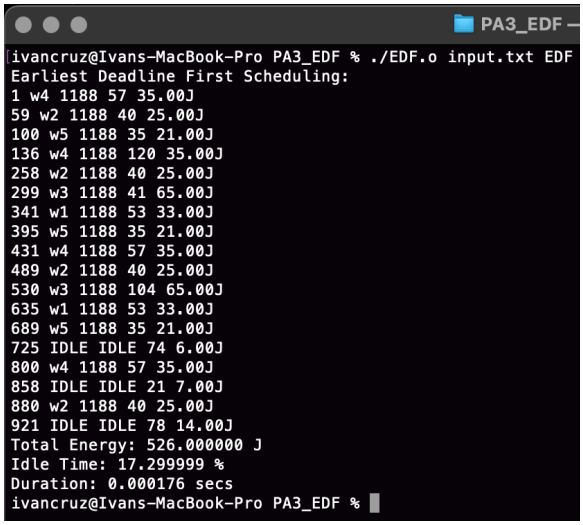

Figure 3: EDF Algorithm for input1.txt

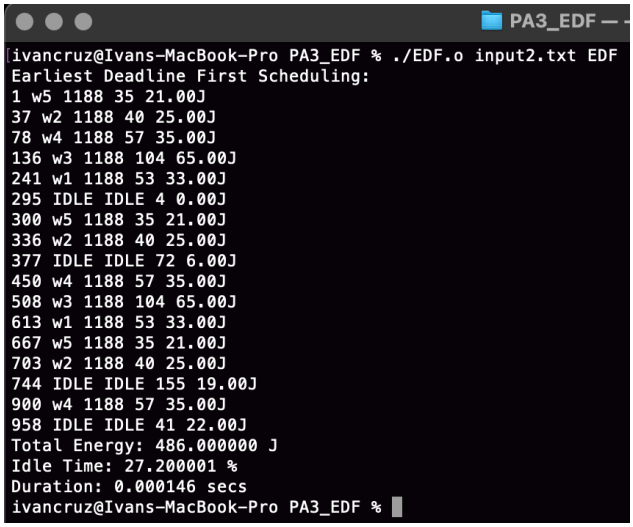

Figure 4: EDF Algorithm for input2.txt

When the EDF algorithm is run with the input files, we can see that it consumes less energy than when it ran with the non-EE RM algorithm. This is because EDF is more preemptive than RM as the priorities for the tasks change more. There is more idle time when compared to the RM counterparts because the required utilization for an EDF schedule is 1 and so there could be more space for the CPU to operate. We could not get to implementing EE EDF but in order to do this, we would have to implement the 5 for loops previously described in the EE RM description. In the last for loop, when we use tTot in the if statement, we will compare it with 1 and not the RM inequality equation. This will grant us a utilization that is both less than and closest to 1. Being closest to 1 allows the schedule to have the least amount of idle time and this helps with energy efficiency. These for loops would be placed at the beginning of the program to determine what executions we would use for each task.