Simon Whitelaw
Operating Systems
Final Project

A Scheduling Algorithm for soft real-time systems with an equal emphasis on process priority and meeting deadlines.

Abstract:
This paper introduces a new Scheduling Algorithm called P-Shift. P-Shift is a dynamic scheduling algorithm for soft real-time operating systems that include processes with and without deadlines such as a laptop. In these types of systems, importance of a process can be measured with two metrics: priority of a task tasks and the need to meet a deadline. Current algorithms tend to give preference, and subsequently more CPU time, to one or the other. P-Shift strikes a balance between the two, sometimes giving CPU time to high priority tasks while at other times giving it to tasks with impending deadlines.

Introduction:
Scheduling algorithms for operating systems is a field of research that has been well developed for many years. This development is continuing as systems become more complex and different systems evolve to have different needs and requirements of a scheduler. For example, a simple machine such as one that is administering drugs to a patient in a hospital has very different scheduling requirements from the scheduler on a personal laptop. This project presents a new scheduling algorithm, P-Shift, which is designed for soft real-time (SRT) systems that have processes both with and without deadlines. Here the scheduler on a laptop computer serves as a motivating example for reasons that follow.
Real-time systems, by definition, involve processes that have a deadline. Specifically, soft deadlines are ones that can be missed without causing physical harm but to the detriment of the output. Sound playing on a computer has a processing deadline in order for a user to hear a song contiguously, but if that deadline is missed the system can carry on with the output slightly degraded. Moreover, the personal computer runs processes that are not constrained by deadlines, but are important nonetheless. Hence the need for a scheduling algorithm that takes into account the priority of processes with and without deadlines.
Two widely used algorithms exist for SRT systems: rate monotonic and earliest deadline first. These have been in existence for decades but variations of them are still used because they are optimal. That is, if there is a way to schedule a real-time system, both of these algorithms will be able to do so. Importantly both of these algorithms are preemptive (a requirement for optimality) which necessitates a priority-based system [6]. Because these two algorithms were designed specifically for scheduling processes that have deadlines, real-time processes are prioritized regardless of their priority relative to non-real-time processes executing in a system. This is the issue that P-Shift aims to address.

There has been some relevant research completed with regard to separating process deadlines from priority of processes. Peha[8] presents an algorithm that schedules processes based on their importance (priority), process time (CPU time needed), ready time (time at which they are ready to run), and deadline (if applicable). The importance acts as a weight, which is applied to completion time and lateness (how much a deadline is missed by) and then minimizes the total weighted lateness and weighted completion time. Although Peha does take into account the priority of all processes, lateness takes precedence over completion times and thus a real-time process will always preempt a non-real-time process regardless of the actual priorities of each.

Similar to Peha, Clark [2] introduces an algorithm that calculates a benefit function based on how much benefit is gained from completing portions of a process. This could potentially be implemented to include processes with deadlines, but in Clark's implementation, the algorithm is designed to "maximize the chance of meeting deadlines," [2(61)] which implies that any process without a deadline will be preempted by one with a deadline. In addition, Clark's method schedules processes statically, where as the current project schedules processes dynamically.

A third algorithm created by Nieh [7], SMART, does recognize the problems that arise when real-time processes always have priority. SMART schedules based on priority first, and then it tries to satisfy real-time deadlines. Each process is split into different queues based on absolute priority. Then value tuples are created which effectively determines a processes delay tolerance and urgency in conjunction with its priority. This algorithm effectively leads to the following hierarchy: priority (implemented with queues) and then real-time constraints within that priority queue. CPU time is subsequently distributed using a pfair algorithm.

The SMART algorithm goes to the other extreme of always giving more time to processes that have higher priority and secondarily considering real-time constraints.

To summarize, current algorithms for SRT systems that also include processes without deadlines as well as priority have tended to emphasize either the time constraints or the priority of the processes. None have been shown to integrate both dynamically. In this paper, I introduce a new algorithm, P-Shift that accomplishes this task. Specifically, the algorithm dynamically schedules a process for every time slot on the CPU. The process is chosen with a simple cost algorithm that assigns a cost value to each process based on priority and time until a deadline, if applicable.

Methodology:
Reflecting on the above problem encourages one to ground his/her ideas with a simple motivating example. Consider these two situations:
Situation 1:

Process	Arrival Time	Duration	Deadline (global time)	Priority
1	0	5	X	1
2	4	3	10	2

Situation 2:

Process	Arrival Time	Duration	Deadline (global time)	Priority
1	0	5	X	1
2	0	3	6	2

Note that the deadline is defined as a global time for simplicity of the simulation. Global here refers to the global time interval that the CPU uses, and it represents the time interval by which the process must be finished executing in to meet its deadline. In the first situation, it would be preferable to complete process 1, with the higher priority before beginning process two, which has lower priority because it will still be able to meet it's deadline. Thus, process 2 should not preempt process 1. In the second situation, if process 2 were to wait for process 1 to finish, it would miss its deadline by 2 time blocks, which could render the computation useless. Further, being priority 2, it is not significantly lower priority than process 1. Thus, process 2 should be able to execute before process 1 or preempt process 1 so that process 2 is able to meet its deadline before allowing process 1 to continue.   In order to accomplish these goals, a new function, called cost, is created that is computed as shown below:

```
int cost(int process){  //could also be a string process name
     int cost = process.priority;
     if(process.deadline!=-1){ //if process has a deadline
          //y gives a measure of how early/late a process will
          //finish
          int y = process.deadline-(time + process.duration);
          if(y<=1)   //if it will, at best, be early by one,
               cost--; //reduce cost by one
          if(y<=0)   //if it will, at best, be on time,
               cost--; //reduce cost by one again
          if(y<=-1)  //if it will, at best, be late by one,
               cost--; //reduce cost by one a third time
     }
     return cost;
```

Notice that the cost is an integer that is initialized as the priority of the process. Next, for processes with a deadline, another integer is calculated that represents the amount of time by which a real-time process would be able to make its deadline if it started executing right away. If the process would make its deadline by 1 time block, the cost is reduced by 1. If the process will exactly meet its deadline, the cost is reduced by 2. Finally, if the process is going to miss its deadline, the cost reduces by 3. The effect of this algorithm is to calculate the cost based on priority and pending deadlines.

Implementation of this idea requires the following steps:

1. The cost of each ready process must be calculated at each time interval. Ready processes are those that have an arrival time before or equal to the global time and have not yet fully executed.
2. The process that gains control of the CPU is simply the one with the lowest cost.
    a. In the event of a cost tie, the process that previously had control of the CPU is allowed to continue.
    b. If both have arrived at the same time and have the same cost function, one is chosen at random to run.

The described algorithm has been implemented as a program in java that takes in processes and prints the execution sequence based on the cost function. Thus far, the implementation is strictly theoretical. Java was used for the theoretical model due to the author's familiarity with the language. In addition, the implementation code is based on a version of the shortest job first algorithm, also written in java [11].

Results:
The tests included the two situations mentioned above, the results of which appear below:

Test 1:
```
Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
a                0               5            -1          1
b                4               3            10          2
The Sequence is:  a  a  a  a  a  b  b  b
Average Wait Time: 1/2
Total Tardiness: 0
```

Test 2:
```
Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
a                0               5                 -1          1
b                0               3                 6           2
The Sequence is:  a  a  a  b  b  b  a  a
Average Wait Time: 6/2
Total Tardiness: 0
```

The execution sequences are as desired where the real-time process does not execute until the high priority one finishes in the first situation, but in the second, as the deadline approaches, process b does in fact preempt process a and is able to finish without missing its deadline. One drawback of the current algorithm seen above is the relatively long wait time that is introduced as a result of the preemption in the middle of process a's execution. This, however, allows the higher priority process to execute for three blocks of time, potentially completing important work in that time.

A second round of tests was conducted to exhibit how differences in priority affect preemption by processes with deadlines (Appendix I). These tests show that as priority differences increase between the process with a deadline and the one without, preemption occurs later and later, eventually resulting in no preemption and the more important task finishes completely. The threshold priority difference for non-preemption is currently 3. The effect is some tardiness as well as a higher portion of the more important task being completed before the task with a deadline.

Further testing was completed in order to compare the performance of the P-Shift Algorithm to EDF and Priority algorithms. The metrics used in the analysis included wait time, tardiness, and priority allocation. Wait time is the amount of time that a process spends ready and waiting to execute. Tardiness is a measure of how much time a process with a deadline completes after that deadline. The tardiness of a process that finishes early or on time is 0. Lastly, priority allocation is a measure of the average priority of the currently executing process at each time interval.

Tests were conducted in two scenarios: a heavy load and light load. The heavy load trials included 10 processes with arrival times between time 0 and 5, durations between 1 and 11, priorities between 1 and 6, and half of the 10 processes with deadlines, which were double the duration from the start time of a process. Light load trials consisted of 10 processes with arrival times between time 0 and 10, durations between 1 and 6, priorities between 1 and 6, and half of the 10 processes with deadlines, which were double the duration from the start time of the process. Both heavy and light loads were both used to simulate the different conditions in which a system could possibly operate. (See appendix IV). It is important to note that the light load is only slightly less busy than the heavy load trials and was used in order to obtain measurements just beyond the EDF's ability to schedule all processes with deadlines to meet their deadlines.

30 trials were run with each scheduling algorithm for each load, and data was collected on average wait times, total tardiness (sum of tardiness of each individual process with a deadline), and priority allocation. It is important to note that in each run of 30 trials, different random numbers were used, so the 30 trials used to test the P-Shift algorithm had different parameters than the 30 trials used to test the EDF algorithm, which were different from the 30 trials used to test the priority algorithm. The wait time and tardiness data was analyzed using simple 2-tailed t-tests while the priority allocation results are shown graphically. The wait time and tardiness results are shown in Table 1 and Table 2, and the priority allocation results are shown in figures 1 and 2.

Table 1: Results of wait time and tardiness tests of P-Shift compared with EDF using t tests.

	Heavy Load			Light Load		
	P-Shift	EDF	P Value	P-Shift	EDF	P Value
Average Wait Time	22.6667	21.7	0.4617	8.2333	8.9	0.4329
Average Tardiness	40.0667	15.9333	4.0901E-07	17.7667	5.1333	6.2336E-07

Table 1 shows that there is no significant difference between the average wait times of the two algorithms in either the heavy load or light load situations. Thus, the P-Shift algorithm does not significantly hinder the overall progress of the processes as compared with the EDF algorithm. There is, however, a significant difference between the tardiness of the two algorithms, which was expected since the P-Shift algorithm takes some emphasis away from the processes with deadlines.
Priority allocation can be viewed as the average priority of the task executed over the 30 trials for each time interval. Figures 3 and 4 show these graphs for the heavy and light load scenarios.

Table 2: Results of wait time and tardiness tests of P-Shift compared with Priority using t tests.

	Heavy Load			Light Load		
	P-Shift	Priority	P Value	P-Shift	Priority	P Value
Average Wait Time	22.6667	24	0.2465	8.2333	9.5667	0.1377
Average Tardiness	40.0667	88.5	1.5441E-08	17.7667	37.9666	1.0963E-06

Table 2 comparing P-Shift algorithm and Priority algorithm show similar results to the comparison between P-Shift and EDF. Once again, there is not significant difference between the average wait times in either scenario demonstrated by the high p values. In addition, there is a significant difference between the average tardiness values in both the heavy and light load situations. This result shows the advantage of the P-Shift Algorithm over the Priority algorithm

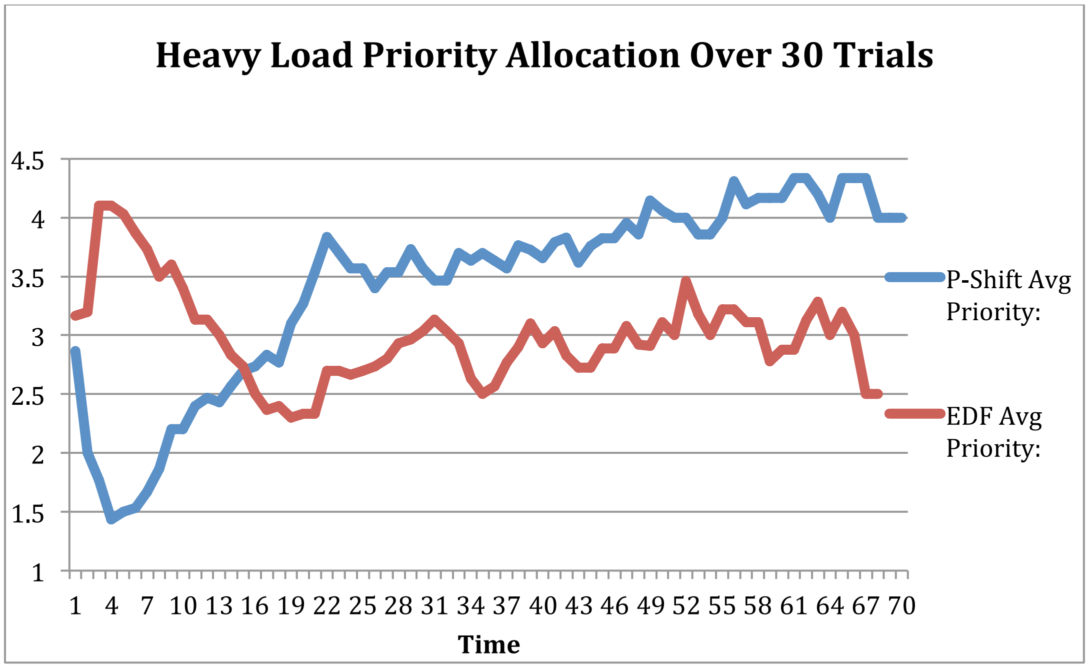

Figure 1: Average priority over 30 trial runs at each time interval for heavy load tests

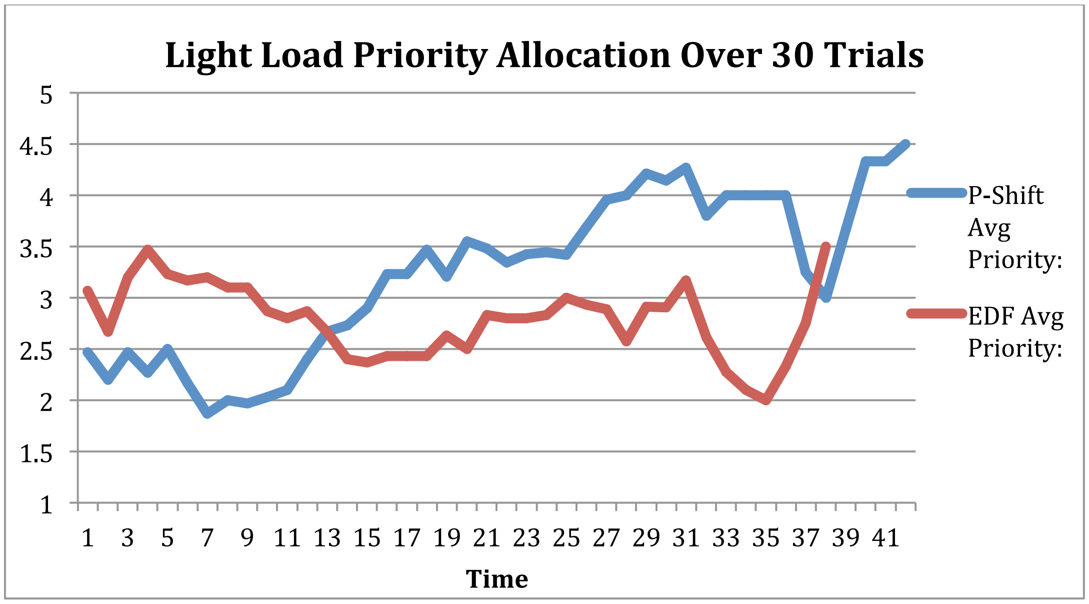

Figure 2: Average priority over 30 trial runs at each time interval for light load tests

Figures 1 and 2 demonstrate the advantage of the P-Shift Algorithm over the EDF algorithm in both the heavy load and light load situations. With the P-Shift algorithm the average priority of the executing processes is consistently lower (more important) at earlier times in the execution while higher priority processes (less important) are executed at later times. The EDF, on the other hand, appears to have more of a flat average priority over time, indicating a lack of emphasis on priority of the processes. Hence, with the P-Shift Algorithm, more important work is given CPU time earlier when compared with the EDF algorithm. Notably, these results do not indicate whether more important processes are being completed earlier, but only that more important work is being completed earlier.

Conclusion and Future considerations:
The results indicate that the algorithm works as desired for a small number of processes. That is, processes gain control of the CPU based on a cost function that depends both on priority and an impending deadline. Here, priority and deadlines are both considered without overemphasizing either one. In particular, with our motivating example, the desired outcome was attained in both situation 1 and 2 by dynamically shifting the cost of the process. In addition, the subsequent analyses show that this algorithm is able to schedule high priority processes to complete work and only inserts lower priority processes when their deadline is approaching. Thus, the algorithm is shown to strike a balance between emphasizing priority and deadlines in a real-time system that also includes tasks without deadlines.
Some areas that this paper does not consider but are crucial to a successful implementation of this algorithm include prevention of starvation, priority inversion, and possible abortion of real-time processes that miss deadlines by a significant margin. Preventing starvation could be accomplished simply by introducing an aging algorithm for processes that are ready but have not been executed. Priority inversion/dependency is another well-known problem that could be handled with existing solutions such as priority inheritance. In order to handle extremely late processes, one could introduce code that dictates whether a process should be aborted based on the amount of time that has passed since it's deadline. Other considerations for future research include an analysis of how much overhead this algorithm introduces when actually implemented in an operating system. This is not a trivial concern due to the dynamic nature of scheduler and the need to calculate the cost for every ready process at every time block. One could also look to tweak the algorithm to either decrease the tardiness of the system or increase the important work that is completed quickly.

Lessons Learned:
This project had three distinct phases, the first of which was the research phase. After researching real-time systems, the initial thought was to complete a project that somehow reduces the latency of an interrupt. This line of thought, however, proved difficult and led to the notion that all the examples were quite limited in that the real time process examples all exclusively contained real time processes. Similarly, I decided that Hard RT systems were too restrictive as well, and subsequently started researching SRT systems. Naturally, this led to numerous papers on SRT schedulers. Much of the current research in the field of SRT schedulers today naturally focuses on multi-core systems. The complicated nature of the field presented challenges in terms of simply reading the papers. In the end, I did find some papers that focused real-time scheduling for single core processors that were from some years ago. In reviewing these older algorithms, I noticed that many of them generally didn't consider real-time and non-real time processes

together, and of those that did, it seemed that none were able to integrate the notion of priority to my satisfaction. The dissatisfaction with the integration of priority and deadlines led to phase two of the project.

Phase two started with the motivating example presented at the beginning of the methodology section. I wanted to create an algorithm that could handle both situations adequately – i.e. scheduling processes according to priority with some way to schedule lower priority processes with deadlines in order that they are able to make (or come close to making) those deadlines. The first attempt involved summations and division, which would have proved much too expensive in terms of overhead for the scheduler. At that point, I landed on the idea of a cost function that acted as a modified priority taking into account the deadline. Once particular challenge was deciding how to modify the cost based on the deadline. Essentially, I didn't want to put too much emphasis on either the priority or the deadline. From that came the adjusted cost based on proximity to the deadline. This phase was also concurrently implemented in java with code based on a SJF scheduler found online.

Phase three included the testing phase. The main challenge here was to come up with a way to methodically test different scenarios and extract that data in a way that could be graphed. Eventually this difficulty was overcome, resulting in the graphs presented in the results section**.**

Appendix: Results of Various Test Runs

I. Tests demonstrating effect of increasing difference in priority – at threshold difference of 3, priority is given full preference over meeting deadline.

```
Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
a                 0                10         -1          2
b                 0                3          6           1
The sequence is:  b  b  b  a  a  a  a  a  a  a  a  a  a
Avaerage Wait Time: 3/2
Total Tardiness: 0

Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
a                 0                10         -1          2
b                 0                3          6           2
The sequence is:  b  b  b  a  a  a  a  a  a  a  a  a  a
Avaerage Wait Time: 3/2
Total Tardiness: 0

Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
a                 0                10         -1          2
b                 0                3          6           3
The sequence is:  a  a  a  b  b  b  a  a  a  a  a  a  a
Avaerage Wait Time: 6/2
Total Tardiness: 0

Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
a                 0                10         -1          2
b                 0                3          6           4
The sequence is:  a  a  a  a  b  b  b  a  a  a  a  a  a
Avaerage Wait Time: 7/2
Total Tardiness: 1

Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
a                 0                10         -1          2
b                 0                3          6           5
The sequence is:  a  a  a  a  a  a  a  a  a  a  b  b  b
Avaerage Wait Time: 10/2
Total Tardiness: 7
```

II. Test demonstrating effective high priority preemption.

```
Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
a                 0                 6          -1          4
b                 0                 2          4           3
c                 4                 2          -1          1
The Sequence is:  b  b  a  a  c  c  a  a  a  a
Avaerage Wait Time: 4/3
Total Tardiness: 0
```

III. Test demonstrating emphasis on high priority over deadline when deadline can easily be made, but emphasizes deadline over priority when the difference in priority is small and the deadline is approaching.

```
Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
a                 0                 10         -1          2
b                 0                 2          4           3
c                 0                 2          -1          1
The Sequence is:  c  c  b  b  a  a  a  a  a  a  a  a  a  a
Avaerage Wait Time: 6/3
Total Tardiness: 0
```

IV. Sample of one of the light load trials used in the 30 trial analysis. Composed of 10 processes with the following attributes: Arrival Time: 0-10; Duration: 1-6; Deadline: half with deadlines approx. double the duration after the arrival time; Priority: 1-6;

```
Process Name|Arrival Time  |  Duration   |   Deadline  | Priority
------------|--------------|-------------|-------------|---------
0                 2                 5          12          4
1                 6                 1          -1          2
2                 8                 1          10          2
3                 9                 2          -1          4
4                 5                 3          11          1
5                 3                 1          -1          5
6                 0                 3          6           5
7                 4                 5          -1          1
8                 5                 5          15          1
9                 7                 2          -1          3
The sequence is:   6  6  0  6  7  8  4  4  4  2  8  8  8  8  0  0  0  0
7  7  7  7  1  9  9  3  3  5
Avaerage Wait Time: 103/10
Total Tardiness: 6
```

References:

[1] Anderson, J. and Srinivasan, A. Pfair Scheduling: Beyond Periodic Task Systems. Seventh International Conference on Real-Time Computing Systems and Applications: 297-306. 2000.

[2] Clark, K. Scheduling Dependent Real Time Activities. Carnegie Mellon University. PhD Dissertation 1990.

[3] Devi, U. Soft Real-Time Schedulers on Multiprocessors. University of North Carolina PhD Dissertation. 2006.

[4] Erickson, J. Managing Tardiness Bounds and Overload in Real Time Systems. University of North Carolina PhD Dissertation 2014.

[5] Erickson, J. and Anderson, J. Reducing Tardiness Under Global Scheduling by Splitting Jobs. 25th Euromicro Conference on Real-Time Systems (ECRTS), 2013: 14-24. 2013.

[6] Mohammadi, A and Akl, S. Technical Report No. 2005-499: Schedulilng Algorithms for Real Time Systems. 2005.

[7] Nieh, J. and Lam, M. A SMART Scheduler for Multimedia Applications. ACM Transactions on Computer Systems: Vol 21: Issue 2: 117-163. 2003

[8] Peha, J. Heteorgeneous-Criteria Scheduling: Minimizing Weighted Number of Tardy Jobs and Weighted Completion Time. Computers and Operations Research: Vol 22: Issue 10: 1089-1100. 1995.

[9] Srinivasan, A. and Anderson, J. Efficient Scheduling of Soft Reat-time Applications on Multiprocessors. 15th Euromicro Conference on Real-Time Systems: 51-59. 2003.

[10] Yang, K. and Anderson, J. Soft Real-Time Semi-Partitioned Scheduling with Restricted Migrations on Uniform Heterogeneous Multiprocessors. Proceedings of the 22nd International Conference on Real-Time Networks and Systems: 215. 2014.

[11] http://www.dreamincode.net/forums/topic/57590-os-scheduling-algorithm-implementation/