九州大学学術情報リポジトリ
Kyushu University Institutional Repository

Evaluation of an Advanced Knowledge-based Process Scheduler

Suranauwarat, Sukanya
Department of Computer Science and Communication Engineering, Graduate School of Information Science and Electrical Engineering, Kyushu University : Graduate Student

Taniguchi, Hideo
Department of Computer Science and Communication Engineering, Faculty of Information Science and Electrical Engineering, Kyushu University

https://doi.org/10.15017/1515733

出版情報：九州大学大学院システム情報科学紀要. 6 (2), pp.179-184, 2001-09-26. 九州大学大学院システム情報科学研究院
バージョン：
権利関係：

KYUSHU UNIVERSITY

九州大学大学院
システム情報科学紀要
第6巻 第2号 平成13年9月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.6, No.2, September 2001

Evaluation of an Advanced Knowledge-based Process Scheduler

Sukanya SURANAUWARAT* and Hideo TANIGUCHI **

(Received June 15, 2001)

Abstract: Conventional process schedulers in operating systems control the sharing of the CPU resources among processes using a fixed scheduling policy, in which the utilization of a computer system (e.g., a real-time or a time-sharing system) is a major concern rather than content or behavior of a process. As a result, the CPU resource is likely to be used in an inefficient manner, or the processing time of a process may be extended unnecessarily. Therefore, we proposed a process' behavior-based scheduler in oder to reduce the processing time and the process switching cost. More specifically, our scheduler allows a process to continue its execution even though its time-slice has already expired, when it is predicted from an advanced knowledge called PFS (Program Flow Sequence) that the process needs a little bit more CPU time before it voluntarily relinquishes the CPU. In this paper, we present the experimental evaluation of our proposed scheduler.

Keywords: Process scheduler, WWW server, Response time, Behavior, Content, Predict

1. Introduction

Conventional process schedulers[1)~4)] in operating systems control the sharing of the CPU resources among processes using a fixed scheduling policy based on the utilization of a computer system such as a real-time or a time-sharing system. Since the control over the allocation of the CPU resource is not based on content or behavior of a process, this can hinder an effective use of a CPU resource or can extend the processing time of a process unnecessarily. For example, in a time-sharing system, when a process uses up its time-slice just before it initiates an I/O operation, it will voluntarily relinquishes the CPU (i.e., the process blocks itself pending the completion of the I/O operation) immediately after the beginning of its next time-slice. If we had predicted the behavior of the process and delayed process switching according to the predicted behavior allowing the process to continue its execution until it initiated an I/O operation, then the processing time of the process and the process switching cost could have been reduced.

Therefore, we have proposed a process' behavior-based scheduler that allows a process to continue its execution even though its time-slice has already expired, when it is predicted from an advanced knowledge called *PFS (Program Flow Sequence)* that the process needs a little bit more CPU time before it voluntarily relinquishes the CPU [5)]. The PFS of each program is created based on the behavior of its corresponding process at the end of the first execution, and it is used whenever the program is executed from then on. It is also adjusted based on the feedback obtained from each execution. We have also implemented the proposed scheduler in BSD/OS 2.1 and evaluated it in simple cases such as the overhead involved in our scheduler [5)]. In this paper, we examine the performance of existing programs when using our scheduler with regard to the length of the time to delay process switching, and the effect on the processing time of adjusting the existing PFS to changes when existing programs are executed a number of times.

2. Overview

In this section, we will briefly describe our scheduler [5)] in order to provide sufficient understanding to the rest of the paper.

When a program is executed, if its PFS does not exist then our scheduler will record the execution behavior of the corresponding process in terms of process identifier and process state (i.e., run, ready, and wait states). And based on this log, the PFS of the program is created at the end of the execution. A PFS is composed of the program name and a sequence of its process information, i.e., a sequence of entries describing process state and time spent. We will refer to each time spent in run state as a CPU time of PFS (T_p).

If the PFS of the program exists, then the decision about whether process switching should be delayed

* Department of Computer Science and Communication Engineering, Graduate Student

** Department of Computer Science and Communication Engineering

or not is determined based on PFS by our scheduler as follows, whenever the corresponding process is running at the end of its time-slice.

- If $T_e - (C_c - C_s) \leq T_m$, then the process is allowed to continue using the CPU.
- If $T_e - (C_c - C_s) > T_m$, then the next waiting process is dispatched.

where C_c is the current time, C_s is the time that a process starts using the CPU for each allocated portion of CPU, T_e is the expected CPU time a process would use from C_s until it voluntarily relinquishes the CPU (each T_e is determined based on each CPU time of PFS (T_p)) and T_m is the maximum time to delay process switching called the *maximum dispatch delay time*.

In other words, whenever a process is running at the end of its time-slice, if the expected CPU time the process would use from now until it voluntarily relinquishes the CPU is smaller than the maximum dispatch delay time (T_m), then we allow the process to continue using the CPU instead of dispatching the next waiting process. We note that setting the T_m arbitrarily will cause the management of process switching to become complex, so we enforce the rule that T_m must be a multiple of *timeslot* where timeslot is the minimum unit of time that process switching can be delayed.

In addtion, our scheduler adjusts PFS to changes based on the feedback obtained from each execution, since the execution behavior of a program is not always the same every time the program is executed. However, adjusting PFS to the latest change is dangerous when the corresponding process runs abnormally. So our scheduler adjusts each CPU time of PFS (i.e., T_p) slightly by multiplying the difference between the CPU time that a corresponding process actually spends before it voluntarily relinquishes the CPU and T_p with a constant (called an *increase* or a *decrease scaling factor*) as shown in the following.

- If $T_p = (C_c - C_s)$, then the adjustment is not needed.
- If $T_p < (C_c - C_s)$, then T_p should be increased by using the following rule:
  $$T_p = T_p + \{(C_c - C_s) - T_p\} \times (x/100), \quad (1)$$
- If $T_p > (C_c - C_s)$, then T_p should be reduced by using the following rule:
  $$T_p = T_p - \{T_p - (C_c - C_s)\} \times (y/100), \quad (2)$$

where x is an increase scaling factor (%) and y is a decrease scaling factor (%).

3. Experimental Evaluation

In this section, we present several experiments designed to evaluate the effectiveness of our scheduler, which is implemented as a modification to the BSD/OS 2.1 kernel. We performed two set of experiments: 1) experiments with a test program, and 2) experiments with existing programs. We chose gzip, merge, and sort as the examples of the existing programs. Gzip is useful for backing-up or transferring large files while merge and sort are used a lot in database systems. The following is describing each program in detail.

A test program is a program that loops 20 times through *work A* and *work B*. Work A increments an integer variable by one for the amount of time specified by the argument sent to the program. Work B goes to sleep in the wait state for a fixed time of 1 s (second). We will refer to the process of the test program as the *test process*.

Gzip is a fast and efficient file compression program distributed by the GNU project. When compressing files, one of the options $-1, -2, \ldots$ through -9 can be used to specify the speed and quality of the compression used. In our experiments, we ran gzip as *"gzip -2 file1"*; file1 contains 200,000 number of integers which are generated by library function called rand().

Merge is a useful program for combining all changes or differences into one file. In our experiments, we ran merge as *"merge outputfile file2 file3"*, resulting all differences that lead from file2 to file3 into outputfile are incorporated. File2 and file3 contain 100,000 number of integers which are generated by rand().

Sort is a text file sorting program. It sorts text files by lines and outputs the results in the standard output or in the file specified by option -o. In our experiments, we ran sort as *"sort file2 file3 -o outputfile"*; file2 and file3 has the same contents as mentioned above.

The purpose of the first set of experiments is to verify that our scheduler works as expected, i.e., it delays process switching as expected, and it adjusts an existing PFS to changes as expected. The purpose of the second set of experiments is to examine the performance of the existing programs when their corresponding processes are scheduled using our scheduler. More specifically, we examined the processing time of each existing program with regard to the length of the maximum dispatch delay time, and the effect on the processing time of adjust-

ing the existing PFS to changes when each program is executed a number of times.

All our experiments were run on a 120 MHz Pentium with 32 MB of memory, running our modified version of BSD/OS 2.1. Also, all experiments were conducted in single user mode with the preemption enabled, and timeslot and time-slice are 1 and 100 ms (milliseconds) respectively. In order to enable process switching when a given time-slice expires, we also ran a CPU intensive program called *loop program*, in every experiments. Loop program is a program that increments an integer variable by one in an infinite loop. We will refer to the process of the loop program as the *loop process*.

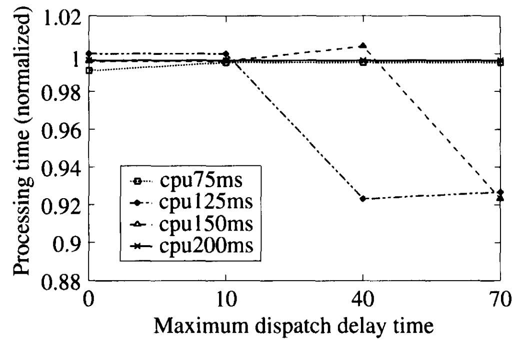

Fig.1 The relation between the length of the maximum dispatch delay time and the processing time when the argument to the test program is given as 75, 125, 150, and 200 ms.

3.1 Test Program

3.1.1 Delay Process Switching

In order to verify that our scheduler can delay process switching as expected, we measured the processing time of the test program when the length of the maximum dispatch delay time was varied as 0, 10, 40, and 70. In this experiment, the argument of the test program was given as 75, 125, 150 and 200 ms. **Figure 1** shows the experimental results plotted with the processing time on y-axis normalized by the processing time when using a conventional time-sharing scheduler. This figure shows that when the required CPU time is more than the time-slice (100 ms) and its difference is smaller than the maximum dispatch delay time, then the processing time when using our scheduler becomes shorter. For example, when the required CPU time is 125 ms and the maximum dispatch delay time is 40 ms, and when the required CPU time is 125 and 150 ms and the maximum dispatch delay time is 70 ms. According to the results, our scheduler delays process switching as expected.

3.1.2 Adjusting An Existing PFS

In order to verify that our scheduler can adapt an existing PFS to changes as expected, we executed the test program with its initial PFS different from its actual execution behavior for 20 times and measured the processing time for each execution. **Figure 2** shows the relation between the number of executions and the processing time, when the maximum dispatch delay time is 20 ms while the increase (x) and the decrease (y) scaling factors are both varied from 10%, 20% and 30%. **Figures 2(a)** and **(b)** show the experimental results when we ran the test program with its argument given as 310 and 110 ms respectively. The initial PFS was created when we ran the test program with its argument given as 210 ms. Also, the processing time in **Fig. 2** is normalized to the processing time when the initial PFS of the test program is the same as the actual execution behavior. Note that during each execution of the test program, CPU time of the PFS is adjusted when the test process goes to sleep 1 s in the wait state.

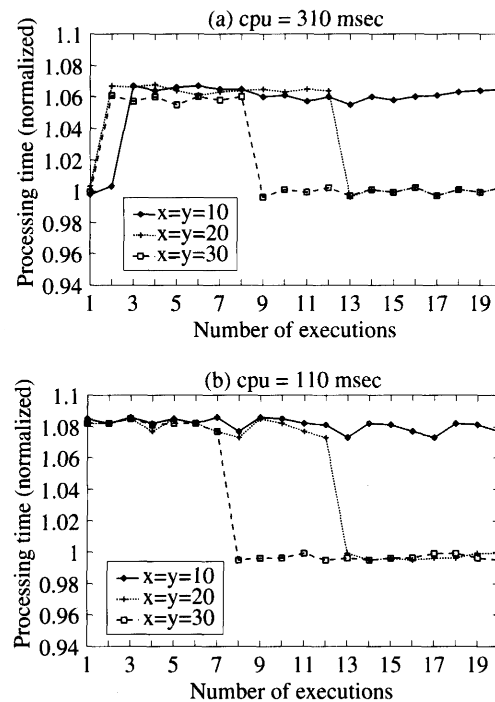

Fig.2 The effect of adjusting the PFS of the test program to changes on its processing time.

During the first execution, when the test process with its argument specifying the amount of time for work A as 310 ms is running at the end of its second time-slice, the CPU time it actually needs before going to sleep 1 s becomes 110 ms, which is more than the maximum dispatch delay time (20 ms). On the other hand, the expected CPU time

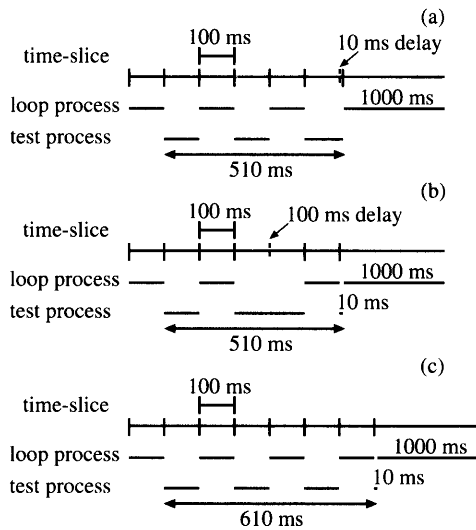

Fig.3 The execution behavior of a test process for work A of each loop when (a) the initial PFS is the same as the actual execution behavior, (b) the number of times the program is executed is small and (c) the CPU time of the initial PFS is adjusted and becomes more than 220 ms.

based on the initial PFS (10 ms) is less than the maximum dispatch delay time (20 ms). Therefore, the initial PFS is adjusted by using (1). That is each CPU time of the initial PFS is increased by using the increase scaling factor. And as the number of executions becomes bigger, it finally becomes close to 310 ms. The experimental results shown in **Fig. 2(a)** are discussed in more detail.

When the number of times the program is executed is small, for example, at the end of the second time-slice of the first execution of the test process, the expected CPU time based on PFS for work A of each loop is 10 ms, which is less than the maximum dispatch delay time (20 ms). As a result, the test process is allowed to continue using the CPU instead of dispatching it to the loop process. After using up 100 ms more of the CPU time, the expected CPU time based on PFS becomes less than zero causing the CPU to be dispatched to the loop process[†1]. According to this, the processing time of the test process when the initial PFS is the same as the actual execution behavior and when it is different are the same as shown in **Figs. 3(a)** and **(b)**. Therefore, the processing time at the first execution in **Fig. 2(a)** is one.

†1 According to our implementation described in 5), the next waiting process is also dispatched when the expected CPU time is less than zero.

As the number of times the program is executed increases, each CPU time of the initial PFS is adjusted and becomes more than 220 ms. Therefore, when the test process is running at the end of its second time-slice, the expected CPU time based on PFS for each loop is more than 20 ms, which is more than the maximum dispatch delay time (20 ms). As a result, the CPU is dispatched to the loop process. Moreover, at the end of its third time-slice, the expected CPU time based on PFS becomes less than zero causing the CPU to be dispatched to the loop process as shown in **Fig. 3(c)**, while the test process with the initial PFS the same as the actual execution behavior is allowed to continue using the CPU for 10 more ms as shown in **Fig. 3(a)**. According to this, the processing time of the test process when the initial PFS is different from the actual execution behavior becomes bigger. Therefore, the processing time in this case is more than one as shown in **Fig. 2(a)**. In addition, **Figure 2**(a) shows that the bigger the increase scaling factor is, the faster each CPU time of PFS becomes more than 220 ms. For example, when the number of executions is two, the processing times with the increase scaling factors of 20% and 30% are more than one, while the one with the increase scaling factor of 10% is still about one.

As the number of times the program is executed increases more, each CPU time of the initial PFS is adjusted and becomes close to 310 ms. Therefore, when the test process is running at the end of its third time-slice, the expected CPU time based on PFS for each loop is less than 20 ms, which is also less than the maximum dispatch delay time (20 ms). As a result, the test process is allowed to continue using the CPU instead of dispatching it to the loop process. According to this, the processing time of the test process when the initial PFS is the same as the actual execution behavior and when it is different become the same. Therefore, the processing time in this case becomes one again as shown in **Fig. 2(a)**. In addition, **Figure 2(a)** shows that the bigger the increase scaling factor is, the faster each CPU time of PFS becomes close to 310 ms. For example, when the number of executions is more than 13, the processing times with the increase scaling factors of 20% and 30% become one again, while the one with the increase scaling factor of 10% is still more than one.

In the same way, the experimental results shown in **Fig. 2(b)** can be explained. In brief, during the first execution, when the test process with its ar-

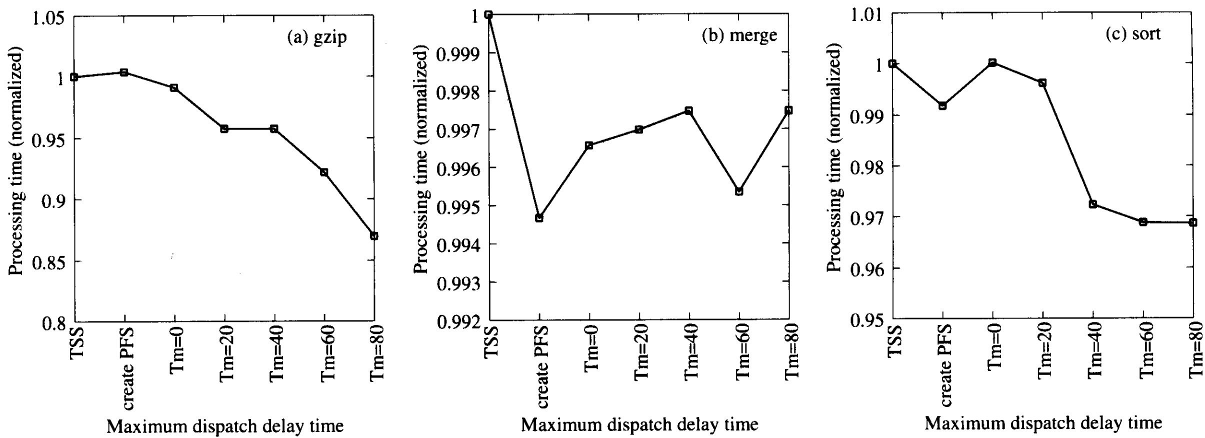

Fig.4 The effect of the maximum dispatch delay time on the processing time (gzip, merge and sort).

gument specifying the amount of time for work A as 110 ms is running at the end of its first time-slice, the CPU time it actually needs before going to sleep 1 s becomes 10 ms while the expected CPU time based on the initial PFS is 110 ms (which is bigger). Therefore, the initial PFS is adjusted by using (2). That is each CPU time of the initial PFS is decreased by using the decrease scaling factor. And as the number of executions becomes bigger, it finally becomes close to 110 ms.

According to the experimental results, our scheduler adjusts an existing PFS to changes as expected.

3.2 Existing Programs

3.2.1 The Length of The Maximum Dispatch Delay Time vs. Processing Time

We ran the three existing programs one at a time and found the relation between the length of the maximum dispatch delay time and the processing time of the process of each program. **Figure 4** shows the experimental results plotted with the processing time on y-axis normalized by the processing time when using a conventional time-sharing scheduler (TSS). For reference, we also show the time used to create PFS in **Fig. 4**. Also, **Table 1** shows, according to PFS, the number of times the CPU and the I/O resources are obtained by each program including time used. Note that the number of times the I/O resource is obtained is shown in parenthesis. Also, the total time used to execute gzip, merge and sort programs are 15, 105 and 19 s respectively. The experimental results shown in **Fig. 4** are discussed in more detail.

In case of gzip: Table 1 shows that the total number of dispatches and the total number of times the CPU resource is obtained by the gzip are

Table 1 The resource usage of gzip, merge and sort analyzed based on PFS.

used time (ms)	number of times CPU (I/O) resources were obtained		
	gzip	merge	sort
under 10	11 (2)	5 (0)	54 (5)
10 to under 20	5 (5)	0 (0)	43 (26)
20 to under 30	0 (7)	0 (1)	12 (33)
30 to under 40	0 (15)	1 (1)	19 (33)
40 to under 50	0 (11)	0 (0)	7 (23)
50 to under 60	0 (7)	0 (0)	1 (16)
60 to under 70	1 (4)	0 (0)	3 (2)
70 to under 80	11 (1)	0 (0)	0 (0)
80 to under 90	0 (0)	0 (0)	0 (4)
90 to under 100	0 (0)	0 (1)	1 (4)
over 100	25 (1)	1 (4)	16 (10)
total	53 (53)	7 (7)	156 (156)

106 and 53 respectively. Out of the total number of times the CPU resource is obtained, about 50% (25 times) of the time it used more CPU time than the time-slice (100 ms). When the required CPU time is more than the time-slice (100 ms), by using our scheduler, the processing time becomes smaller as the length of the maximum dispatch delay time is increased. For example, the processing time when the maximum dispatch delay time is 20 and 60 ms, is improved 4.2% and 7.8% respectively. Therefore, our scheduler makes a noticeable improvement.

In case of merge: Table 1 shows that the total number of dispatches and the total number of times the CPU resource is obtained by the merge are 14 and 7 respectively. Out of the total number of times the CPU resource is obtained, only one time did it use more CPU time than the time-slice (100 ms). As a result, the processing time when

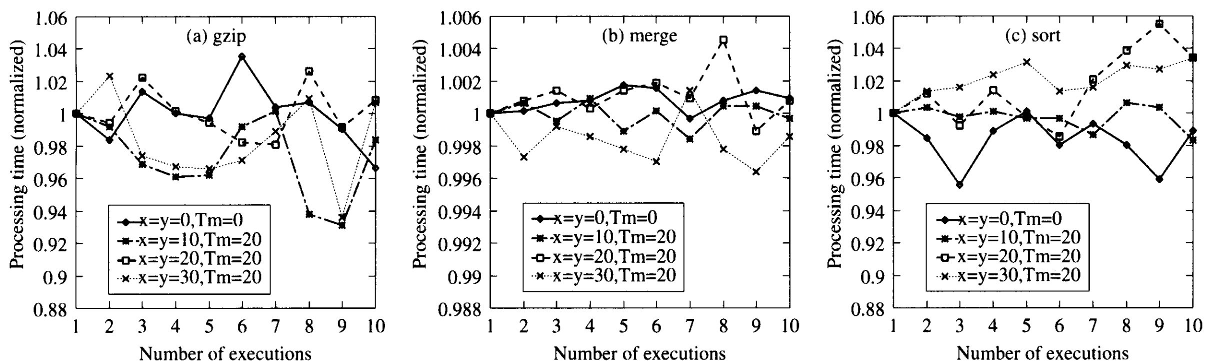

Fig.5 The effect of adjusting the existing PFS to changes on the processing time (gzip, merge and sort).

using our scheduling is almost the same as when not using it, regardless of how much the length of the maximum dispatch delay time is increased. For example, the processing time when the maximum dispatch delay time is 60 ms is improved only 0.5%. Therefore, the effect of our scheduler is relatively small in this case.

In case of sort: Table 1 shows that the total number of dispatches and the total number of times the CPU resource is obtained by the sort are 312 and 156 respectively. Out of the total number of times the CPU resource is obtained, only 10% (16 times) of the time it used CPU time more than the time-slice (100 ms). However, the processing time becomes smaller as the length of the maximum dispatch delay time is increased. For example, the processing time when the maximum dispatch delay time is 60 ms is improved 3.0%. In this case also, our scheduler makes a noticeable improvement.

3.2.2 Adjusting An Existing PFS vs. The Processing Time

We varied the number of executions of each program from 1 to 10 and observed the effect on the processing time of adjusting the existing PFS. **Figure 5** shows the relation between the number of executions and processing time, when the maximum dispatch delay time is 20 while the increase (x) and the decrease (y) scaling factors are both varied from 10%, 20% and 30%. For comparison, we also show the results when the maximum dispatch delay time and the increase and the decrease scaling factors are zero. In addition, the processing time in **Fig. 5** is normalized to the processing time of the first execution. Note that during each execution of the existing programs, CPU time of the PFS is adjusted when the processes of the existing programs are blocked for an I/O operation to be completed in the wait state.

In **Fig. 5**, there is a trend that the processing time of the process of gzip program will decrease as the number of executions becomes larger, while we did not notice this kind of trend for the merge and the sort programs. This shows that adjusting the PFS of the three existing programs has very little or no effect on the processing time.

4. Conclusions

In this paper, we evaluated the effectiveness of our proposed scheduler experimentally. Our results with the test program show that our scheduler works as expected. That is, it delays process switching and adjusts an existing PFS to changes as expected. Also, the results with the existing programs (i.e., gzip, merge, and sort) show that the processing time of the corresponding process of each program can be reduced by using our scheduler.

Some of our future work will include evaluating the effectiveness of our scheduler with other existing programs, and implementing early process switching.

References

1) A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," IEEE Trans. Software Eng., 21(5):475–479, 1995.
2) W. Feng and J. Liu, "Algorithms for scheduling real-time tasks with input error and end-to-end deadlines," IEEE Trans. Software Eng., 23(2):93–106, 1997.
3) B. Ford and S. Susarla, "CPU inheritance scheduling," Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation, pp.91–106, 1996.
4) P. Goyal, X. Guo, and H. Vin, "A hierarchical CPU scheduler for multimedia operating systems," Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation, pp.107–121, 1996.
5) S. Suranauwarat and H. Taniguchi, "The design and implementation of an advanced knowledge-based process scheduler," Research Reports on Information Science and Electrical Engineering of Kyushu University, 6(1):25–30, 2001.