The Design, Implementation and Initial Evaluation of an Advanced Knowledge-based Process Scheduler

Sukanya Suranauwarat Hideo Taniguchi

Graduate School of Information Science and Electrical Engineering,
Kyushu University, Fukuoka-shi, 812-8581, Japan
{sukanya,tani}@csce.kyushu-u.ac.jp

Abstract Traditional operating systems control the execution of programs regardless of how often they are run. This raises the question: can't the often run or the often used programs provide better performance if an operating system had an ability to optimize their execution behavior based on a knowledge the operating system had obtained from their previous execution(s)? In this paper, we integrate this ability into a part of an operating system called a process scheduler and examine its cost and benefit. Our initial evaluations show that the cost involved in our scheduler is small and the processing time can be reduced by using this scheduler.

1. Introduction

Traditional operating systems control the execution of programs regardless of how often they are run. In other words, with traditional operating systems, the often used programs are treated in the same way as those that are rarely run or those that are run for the first time. However, among the programs that are in execution, it is desirable for users to have their often used programs to be executed at the highest speed.

Therefore, in this paper, we ask whether the often used programs might provide better performance if an operating system had an ability to optimize or to alter their execution behavior based on an advanced knowledge — a knowledge the operating system obtained from the previous execution(s) of the programs. So, we decided to integrate this ability into a part of an operating system called a process scheduler, and we design a scheduler that alters the execution behavior of a program by changing the timing of process switching. More specifically, our scheduler delays process switching in order to allow a process running on behalf of an often used program to continue its execution even though its time-slice has already expired, when it is predicted from an advanced knowledge that that process needs a little bit more CPU time before it voluntarily relinquishes the CPU. By doing this, we can achieve enhanced performance.

As a concrete example, by delaying process switching of an objective process that uses up its time-slice just before it initiates an I/O operation, its processing time and process switching cost can be reduced. Since it does not need to wait until its next time-slice to initiate an I/O operation and relinquish the CPU

immediately after that for the I/O completion. As mentioned, the decision about whether or not to delay the process switching is determined based on an advanced knowledge, which we will call *PFS (Program Flow Sequence)*. The PFS of each program is created based on the execution 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 implement our scheduler in BSD/OS 2.1 and evaluate its effectiveness experimentally. Our initial experimental results show that the cost involved in our scheduler is small and the processing time can be reduced by using our scheduler.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 and Section 4 describe the design and implementation of our scheduler respectively. Section 5 describes our experiments and explains the results we obtained. Section 6 offers our conclusions and future work.

2. Related Work

The work described in this paper relates mainly to the area of CPU scheduling in operating systems.

Traditional process schedulers 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. Real-time system schedulers are usually only available in real-time operating systems, and not in general-purpose operating systems in which time-sharing system schedulers are used. In other words, real-time and time-sharing system schedulers have existed in two separate worlds. However, the advent of multimedia applications on PCs and workstations has called for new scheduling paradigms to support real-time in systems with conventional time-sharing schedulers. One simple approach to do this, which has been adopted by many general-purpose operating systems such as Solaris, Linux and Windows NT, is to provide fixed priorities to real-time applications [1][2]. Another approach is to schedule based on proportion and/or period [3][4][5][6]. Another approach is based on hierarchical scheduling with several scheduling classes and with each application being assigned to one of these classes for the entire duration of its execution [7][8][9].

However, none of the above approaches is trying to schedule based on content or behavior of a process, which is what our approach does. As a consequence, in some cases, this can hinder an effective use of the CPU resource or can extend the processing time of a process unnecessarily.

For example, in a time-sharing system, if the process is still running at the end of its time-slice, the CPU is preempted and given to the next waiting process no matter how much more CPU time the process needs. Thus, a process that needs just a little bit more CPU time will also need to wait until its next time-slice. Because of this, the processing time and the process switching cost increase unnecessarily. For example,

when a process uses up its time-slice just before it initiates an I/O operation, it will voluntarily relinquish 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 extra costs mentioned above would have been avoided.

In a parallel computer system, scheduling a parallel program onto CPUs is basically done based on behavior of processes by taking a directed acyclic graph representing the execution behavior of the parallel program (e.g., dependencies of code segments) as input, and schedule it onto CPUs of a target machine in a manner which reduces the completion time [10][11][12]. However, scheduling is performed without regard to the previous execution behavior of the parallel program as our approach does.

In addition to process scheduling, one function in Windows 98, which users can get “faster program start up” as a performance enhancement [13], uses an idea similar to ours. That is, the function improves the performance of a user's programs based on the previous usage of the programs. To be more specific, the function creates a log file to determine which programs a user runs most frequently. All such frequently used files are then placed in a single location on the user's hard disk, which further reduces the time needed to start those programs [14]. However, the function does not alter the execution behavior of programs based on the previous usage, which is what our idea does. Therefore, by using the function in Windows 98, the operating system can control a user's programs more efficiently until a user's programs start up (i.e., the operating system can locate and load a user's programs faster), but it cannot execute or run a user's programs more efficiently.

3. Design

Our idea-based scheduler can be divided in two parts: the *logger* and the *process controller*. When a program is executed, if its PFS does not exist then the logger will record the execution behavior of the corresponding process and use it to create PFS of the program at the end of the execution. If its PFS exists then the process controller will alter the execution behavior of the corresponding process based on PFS, which in this paper means that it will make a decision about whether or not the process switching should be delayed based on the PFS, when the corresponding process is running at the end of its time-slice. The process controller also adjusts PFS to changes in execution behavior of a program. Note that, in this paper, we discuss only the programs that consist of a single process in which the mutual relation between processes in the same program is not a concern.

Figure 1 is a simple example used to identify how our scheduler works and how it improves the performance. In this figure, process A and process B need respectively 3.4 seconds and 2.1 seconds of CPU

time to complete their jobs. Both processes have the same priority and a time-slice of 1 second. Figures 1(a) and (b) show the processing times of process A and process B when using a conventional timesharing scheduler and when using our scheduler respectively. In Figure 1(a), the processing times of process A and process B are 5.5 seconds and 4.1 seconds respectively, while in Figure 1(b), based on the advanced knowledge, PFS, that process B needs only 0.1 seconds more CPU time to complete its job, the scheduler delays process switching by 0.1 seconds to allow process B to complete its job. Delaying process switching causes the processing time of process B to be reduced to 3.1 seconds while that of process A is still the same as in Figure 1(a). Moreover, the number of process switches of the process B decreases from 6 to 4.

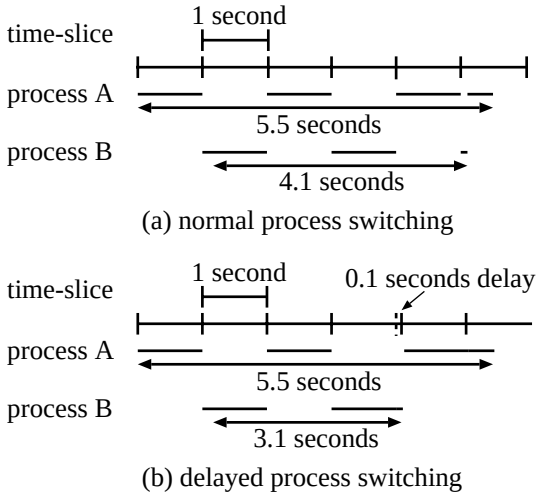

Figure 1: A scheduling example.

The following sections discuss the parts of our scheduler in more detail.

3.1 The Logger

When a program is executed, if its PFS does not exist then a log about the corresponding process is collected recording the information necessary to create PFS. A log (shown in Figure 2(a)) is a sequence of entries describing time, process identifier and process state. This information is recorded at dispatch time, i.e., when deciding which process to run next. Note that there are many process states, but we will focus only on run, ready, and wait states. A process is said to be running in the run state if it is currently using the CPU. A process is said to be ready in the ready state if it could use the CPU if it were available. A process is said to be blocked in the wait state if it is waiting for some event to happen (e.g., an I/O completion event) before it can proceed.

Next, using the log mentioned above creates PFS. A PFS (shown in Figure 2(b)) 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 the run state as a CPU time of PFS (T_P).

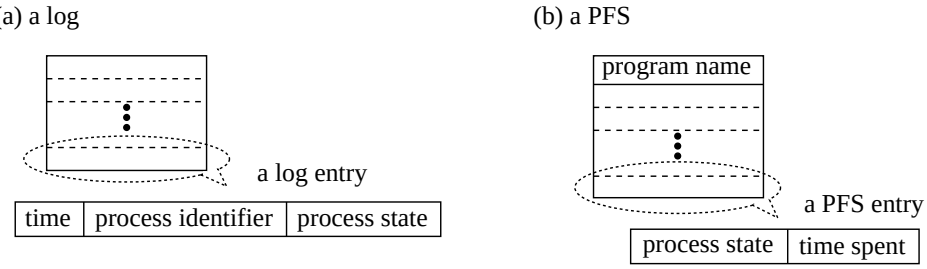

Figure 2: The image of a log and a PFS.

3.2 The Process Controller

When a process is running at the end of its time-slice, if the PFS of the program it runs on behalf of exists, then the decision about whether process switching should be delayed or not is determined based on PFS as follows.

- If $T_e - (C_c - C_s) \le T_m$, then the process is allowed to continue using the CPU, (1)
- If $T_e - (C_c - C_s) > T_m$, then the next waiting process is dispatched, (2)

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 *maximum dispatch delay time*.

According to this, when 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.

Since the execution behavior of a program may change due to, for example, the load in the system, PFS needs to be able to adjust to changes. However, adjusting PFS to the latest change is dangerous when the corresponding process runs abnormally. Therefore, the process controller adjusts each CPU time of PFS (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. (3)
- If $T_P < (C_c - C_s)$, then T_P should be increased by using the following rule:

  $$T_P = T_P + \{(C_c - C_s) - T_P\} \times (x/100), \quad (4)$$

  where x is an increase scaling factor (%).
- If $T_P > (C_c - C_s)$, then T_P should be reduced by using the following rule:

  $$T_P = T_P - \{T_P - (C_c - C_s)\} \times (y/100), \quad (5)$$

  where y is a decrease scaling factor (%).

According to this, throughout the execution, whenever a process voluntarily relinquishes the CPU (e.g., when the process blocks itself pending the completion of the I/O operation in the wait state), if T_P is smaller or greater than the CPU time that the process actually spends before it voluntarily relinquishes the CPU, then it is increased or decreased slightly by using an increase or a decrease scaling factor.

4. Implementation

This section describes the implementation of the logger and the process controller in detail.

4.1 The Logger

The main work of the logger is to create a log and a PFS. A log (shown in Figure 3) is implemented as an array of structures containing information about time (*clock*), process identifier (*pid*) and process state (*p_state*). A PFS (shown in Figure 3) is implemented as an array of structures containing information about process state (*p_state*) and time spent (T_P). In order to identify the PFS of each program, the *program name* is attached to the top of the array. Figure 3 shows how to use the log to create a PFS and it is described in detail below.

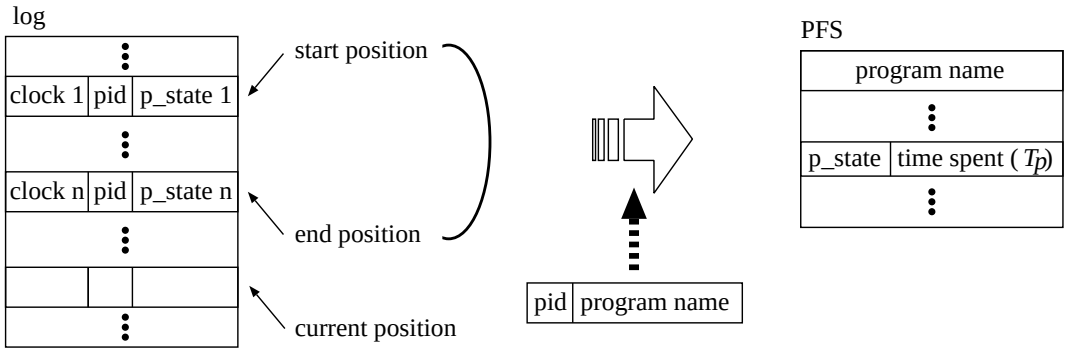

Figure 3: The diagram of how to use the log to create a PFS.

How to create a log: when a program is executed, if its PFS does not exist then the information necessary to create the PFS is recorded in the log at every dispatch until the corresponding process terminates. That is,

- when the corresponding process is the current running process, the information about time, process identifier and process state from now (i.e., ready or wait state) is entered into the log via a pointer giving the current position. Then, the pointer is incremented to the next log entry and the next waiting process is dispatched.
- when the corresponding process is the next waiting process, the information about time, process identifier and process state from now (i.e., run state) is entered in the log via a pointer giving the current position. Then, this pointer is incremented to the next log entry and the corresponding process is dispatched.

Note that the addresses of the log entry when the process starts execution and when it finishes execution are respectively referred to as the *start* and the *end positions*. And the data between these two positions are used for creating the PFS of the program.

How to create a PFS: at the end of the execution, space for the PFS of the program is allocated and the data for each PFS entry is then created from the log between the start and the end positions. When there is more than one process in the system, it is necessary to determine which process(es) correspond to which program. By using this information, the data for the process(es) which correspond to the program name is taken from the log. Note that for our scheduler only the time spent in run state and in wait state have useful information, since the time spent in ready state depends on the number of the processes waiting for the CPU to become available in the ready queue and has no bearing on future execution behavior. Therefore, we consider the series of log entries in which their *p_state* element recording the consecutive switching

between run state and ready state, as one PFS entry whose *p_state* element is run state and T_P element is the summation of the times spent in run state in that series, and it is calculated using the following equation where C_i is a *clock i* element in the log entries in that series.

$$T_p = \{C_2(ready) - C_1(run)\} + \cdots + \{C_n(ready) - C_{n-1}(run)\} + \{C_{n+2}(wait) - C_{n+1}(run)\}$$

4.2 The Process Controller

The main work of the process controller is to make a scheduling decision based on PFS and to adjust an existing PFS to changes. This can be implemented by dividing into four phases: 1) when a process is created, 2) when a process uses up its time-slice, 3) when a process blocks itself, and 4) when a process terminates. The following is describing each phase in more detail. The diagram of how to schedule based on PFS and the processing flowchart are shown in Figures 4 and 5 respectively.

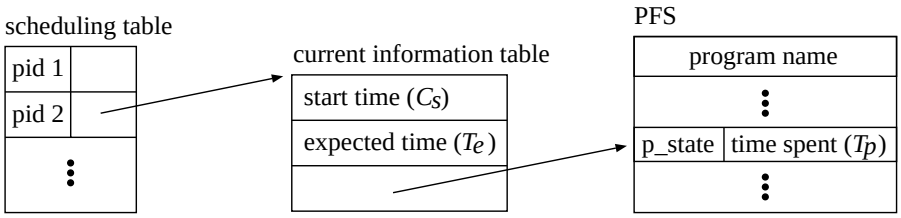

Figure 4: The diagram of how to schedule a process based on PFS.

1) When a process is created,

if the PFS of the program it run exists, then

(a) its process identifier is stored in a table called *scheduling table* and space for a table called *current information table* is allocated. A scheduling table (shown in Figure 4) is an array of structures containing process identifier (*pid*) of the process that will be scheduled based on PFS and a pointer to its current information table. A current information table (shown in Figure 4) is a structure containing the time a process starts using the CPU for each allocated portion of CPU (C_s), the expected CPU time a process would use from C_s until it voluntarily relinquishes the CPU (T_e) and a pointer to the PFS entry (*pfs_ptr*).

(b) Next, each element of the current information table is initialized as below.

- $C_s \leftarrow 0$,

- *pfs_ptr* $\leftarrow$ the address of the first PFS entry whose *p_state* is run state,
- $T_e \leftarrow T_P$ which is accessed via *pfs_ptr*.

2) When a process uses up its time-slice,

(a) the elements of the current information table, T_e and C_s, are updated as below.

- $T_e \leftarrow T_e - (C_c - C_s)$,
- $C_s \leftarrow C_c$ if $T_e \leq T_m$, or

  $C_s \leftarrow 0$ if $T_e > T_m$,

  where T_m is the maximum dispatch delay time.

(b) Next, the process is scheduled according to (1) and (2). That is,

- if $T_e \leq T_m$, then the process is allowed to continue using the CPU.
- if $T_e > T_m$, then the next waiting process is dispatched.

  Note that the next waiting process is also dispatched if $T_e < 0$.

3) When a process blocks itself,

(a) the element of PFS entry, T_P, is adjusted according to (3) through (5). That is,

- if $T_e - (C_c - C_s) = 0$, then $T_P \leftarrow T_P$ (no adjustment).
- if $T_e - (C_c - C_s) < 0$, then $T_P \leftarrow T_P - \{T_e - (C_c - C_s)\} \times (x/100)$
- if $T_e - (C_c - C_s) > 0$, then $T_P \leftarrow T_P - \{T_e - (C_c - C_s)\} \times (y/100)$

(b) Each element of the current information table is updated as below.

- $C_s \leftarrow 0$,
- *pfs_ptr* $\leftarrow$ the address of the next PFS entry whose *p_state* is run state,
- $T_e \leftarrow T_P$ which is accessed via *pfs_ptr*.

4) When a process terminates,

(a) the element of PFS entry, T_P, is adjusted according to (3) through (5). That is,

- if $T_e - (C_c - C_s) = 0$, then $T_P \leftarrow T_P$ (no adjustment).
- if $T_e - (C_c - C_s) < 0$, then $T_P \leftarrow T_P - \{T_e - (C_c - C_s)\} \times (x/100)$

- if $T_e - (C_c - C_s) > 0$, then $T_P \leftarrow T_P - \{T_e - (C_c - C_s)\} \times (y/100)$

(b) The current information table is free and its associated scheduling table entry is removed.

Note that when a process changes from ready state to run state, if $C_s = 0$ then $C_s \leftarrow C_c$.

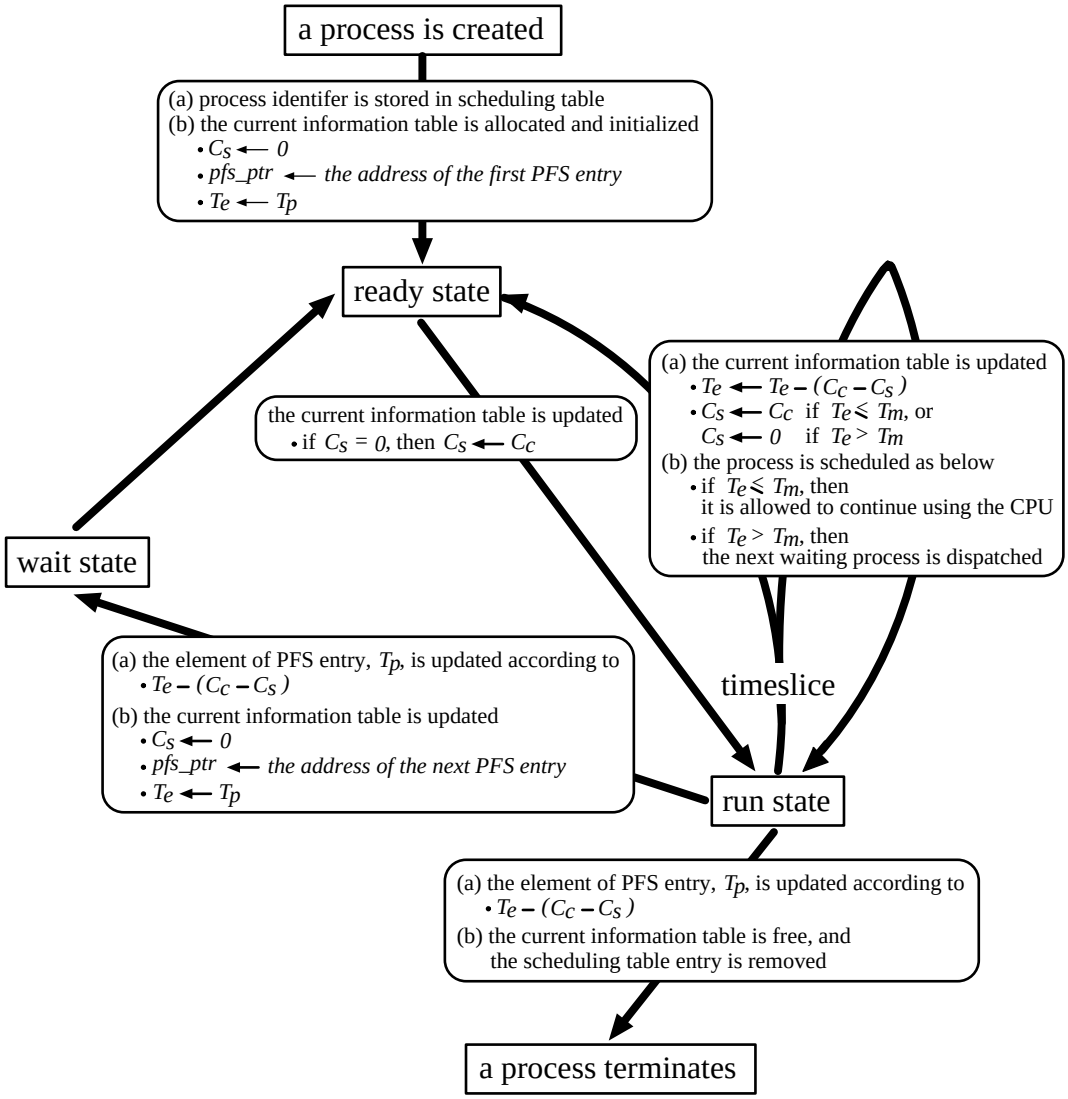

Figure 5: The processing flowchart.

5. 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 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. The basic function of gzip is to take a file, compress it, save the compressed version as filename.gz, and remove the original, uncompressed file. When compressing files, one of the options $-1,-2,\cdots,-9$ can be used to specify the speed and quality of the compression used. -1 specifies the fastest method, which compresses the files less compactly, while -9 uses the slowest but best compression method. In our experiments, we ran gzip as "gzip –2 file1"; file1 contains 200,000 number of integers which are generated by the 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 explore the cost involved in our scheduler, and 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 adjusting 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. In addition, the experiments were conducted in single user mode with the preemption enabled, and timeslot and time-slice are 1 and 100 milliseconds respectively.

5.1 Test Program

This section presents the experimental results with the test program for the cost involved in our scheduler and the validation of our scheduler.

5.1.1 Overhead

The cost involved in creating and storing PFS on memory and the cost involved in scheduling based on PFS, when running the test program, are as follows.

- We did not notice a difference between the processing time when a PFS was created for the test process and when it was not. Therefore, the overhead of logging the information of the test process and creating PFS is relatively small.
- When the maximum dispatch delay time is zero, the processing time when the test process is scheduled by our scheduler is the same as when it is scheduled by a conventional time-sharing scheduler. Therefore, the overhead of scheduling based on PFS is small.

According to the above experimental results, the overhead of our scheduler is small.

5.1.2 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 milliseconds. In order to enable process switching when a given time-slice expires, the test program coexisted with a CPU intensive program called *loop program*, throughout the experiment. 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*. Figure 6 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 milliseconds) 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 milliseconds and the maximum dispatch delay time is 40 milliseconds, and when the required CPU time is 125 and 150 milliseconds and the maximum dispatch delay time is 70 milliseconds. According to the results, our scheduler delays process switching as expected.

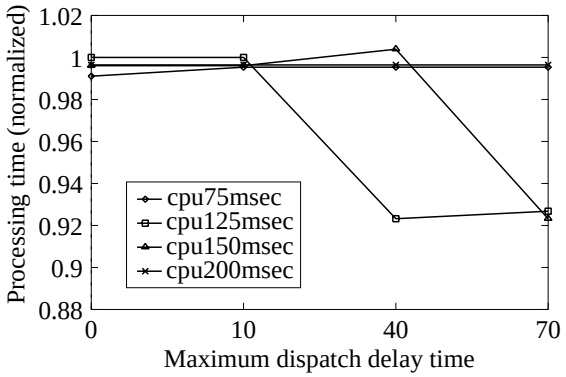

Figure 6: The relation between the maximum dispatch delay time and the processing time of the test program when the argument to the test program is given as 75, 125, 150, and 200 milliseconds.

5.1.3 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. During the experiment, the test program coexisted with the loop program, in order to enable process switching when a given time-slice expires. Figure 7 shows the relation between the number of executions and the processing time, when the maximum dispatch delay time is 20 milliseconds while the increase (x) and the decrease (y) scaling factors are both varied from 10%, 20% and 30%. Figures 7(a) and (b) show the experimental results when we ran the test program with its argument given as 310 and 110 milliseconds respectively. The initial PFS was created when we ran the test program with its argument given as 210 milliseconds. Also, the processing time in Figure 7 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 second in the wait state.

During the first execution, when the test process with its argument specifying the amount of time for work A as 310 milliseconds is running at the end of its second time-slice, the CPU time it actually needs before going to sleep 1 second becomes 110 milliseconds, which is more than the maximum dispatch delay time (20 milliseconds). On the other hand, the expected CPU time based on the initial PFS (10 milliseconds) is less than the maximum dispatch delay time (20 milliseconds). Therefore, the initial PFS is adjusted by using (4). 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 milliseconds. The experimental results shown in Figure 7(a) are discussed in more detail.

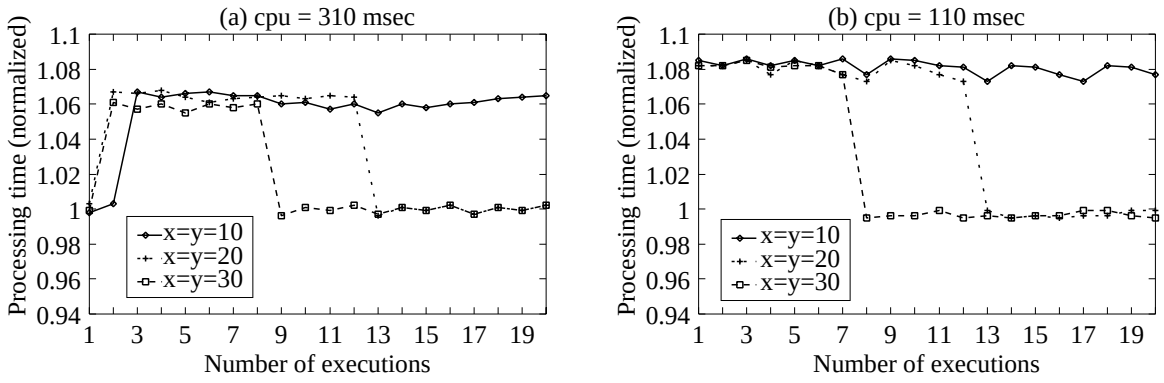

Figure 7: The effect of adjusting the PFS of the test program to changes on its processing time.

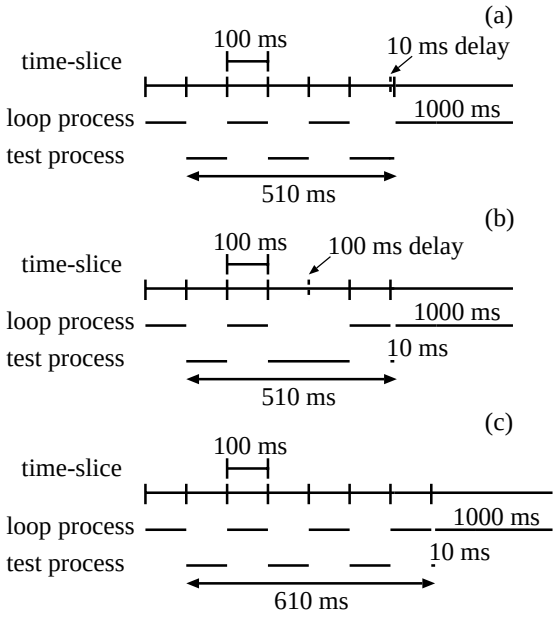

Figure 8: The execution behavior of a test process for work A of each loop when (a) the initial PFS is the same as the actual execution behavior, (b) the number of times the program is executed is small, and (c) the CPU time of the initial PFS is adjusted and becomes more than 220 milliseconds.

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 milliseconds, which is less than the maximum dispatch delay time (20 milliseconds). 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 milliseconds 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 (recall the rule (in Section 4.2) that the next waiting process is dispatched if $T_e < 0$). 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 Figures 8(a) and (b). Therefore, the processing time at the first execution in Figure 7(a) is one.

As the number of times the program is executed increases, each CPU time of the initial PFS is adjusted and becomes more than 220 milliseconds. 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 milliseconds, which is more than the maximum dispatch delay time (20 milliseconds). 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 Figure 8(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 milliseconds as shown in Figure 8(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 Figure 7(a). In addition, Figure 7(a) shows that the bigger the increase scaling factor is, the faster each CPU time of PFS becomes more than 220 milliseconds. 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 milliseconds. 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 milliseconds, which is also less than the maximum dispatch delay time (20 milliseconds). 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 Figure 7(a). In addition, Figure 7(a) shows that the bigger the increase scaling factor is, the faster each CPU time of PFS becomes close to 310 milliseconds. 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 Figure 7(b) can be explained. In brief, during the first execution, when the test process with its argument specifying the amount of time for work A as 110 milliseconds is running at the end of its first time-slice, the CPU time it actually needs before going to sleep 1 second becomes 10 milliseconds while the expected CPU time based on the initial PFS is 110 milliseconds (which is bigger). Therefore, the initial PFS is adjusted by using (5). 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 milliseconds.

5.2 Existing Programs

This section shows the effect of the maximum dispatch delay time and the effect of adjusting the PFS of each existing program (gzip, merge, and sort), on its processing time.

5.2.1 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. In order to enable process switching when a given time-slice expires, throughout the experiment, the existing programs coexisted with the loop program. Figure 9 shows the experimental results plotted with the processing time on y-axis normalized by the processing time when using a conventional timesharing scheduler (TSS). For reference, we also show the time used to create PFS in Figure 9. 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 seconds respectively. The experimental results shown in Figure 9 are discussed in more detail.

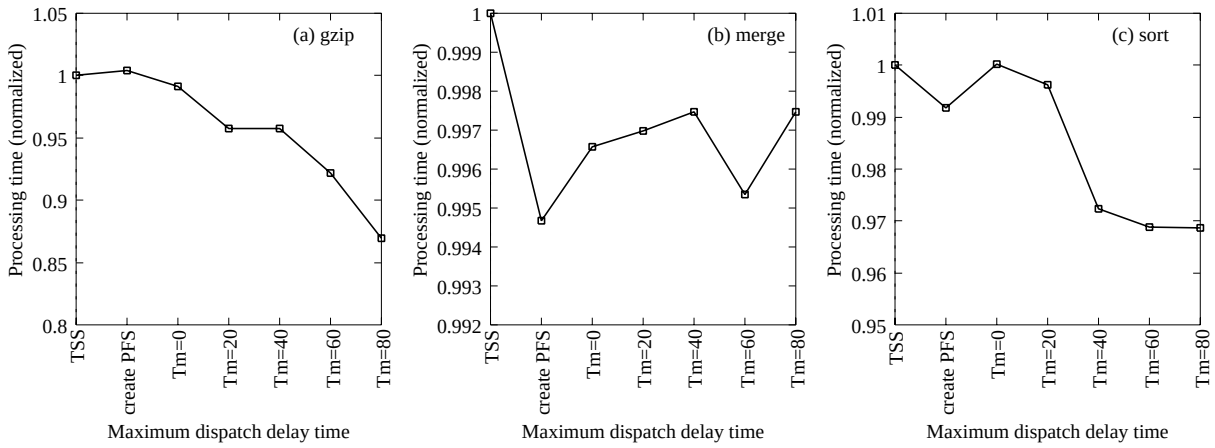

Figure 9: The effect of the maximum dispatch delay time on the processing time (gzip, merge, and sort).

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

time used (milliseconds)	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)	total 7 (7)	total 156 (156)

In the case of gzip: table 1 shows that the total number of dispatches and the total number of times the CPU resource is obtained by gzip are 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 milliseconds). When the required CPU time is more than the time-slice (100 milliseconds), 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 milliseconds, is improved 4.2% and 7.8% respectively. Therefore, our scheduler makes a noticeable improvement in this case.

In the case of merge: table 1 shows that the total number of dispatches and the total number of times the CPU resource is obtained by 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 milliseconds). And that time it used the CPU resource only 3 milliseconds longer than its normal time-slice. As a result, the processing time when 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 milliseconds is improved only 0.5%. Therefore, the effect of our scheduler is relatively small in this case.

In the case of sort: table 1 shows that the total number of dispatches and the total number of times the CPU resource is obtained by 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 milliseconds). 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 milliseconds is improved 3.0%. In this case also, our scheduler makes a noticeable improvement.

5.2.2 Adjusting An Existing PFS vs. 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. During the experiment, the existing programs coexisted with the loop program, in order to enable process switching when a given time-slice expires. Figure 10 shows the relation between the number of executions and processing time, when the maximum dispatch delay time is 20 milliseconds 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 Figure 10 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 Figure 10, 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.

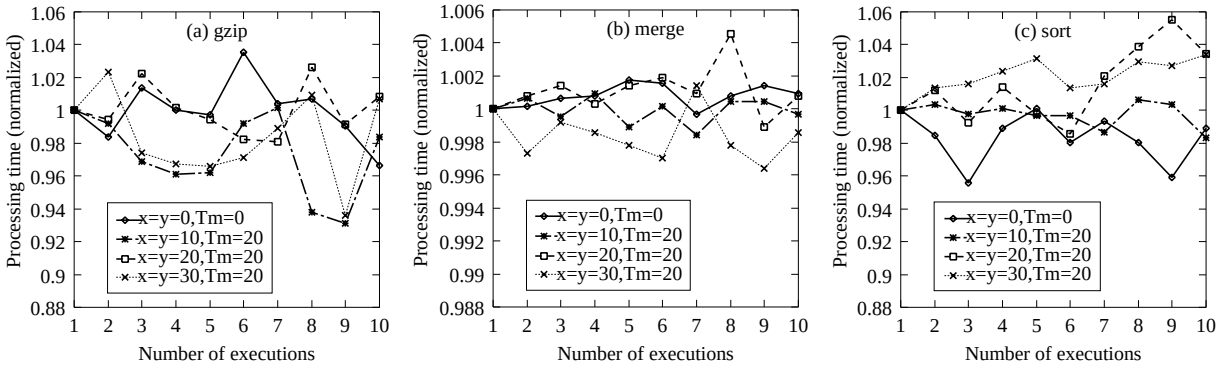

Figure 10: The effect of adjusting the existing PFS to changes on the processing time (gzip, merge, and sort).

6. Conclusions

This paper proposed a process scheduler that controls the sharing of the CPU resource based on behavior of a process. The special feature of our scheduler is that (1) when a program is executed for the first time, it

logs the behavior of the corresponding process at every dispatch and then creates an advanced knowledge called PFS (Program Flow Sequence) at the end of the execution, and (2) when the program is executed from then on, it schedules the corresponding process based on the PFS, i.e., it delays process switching in order to allow the corresponding process to continue its execution, when it is predicted from PFS that the corresponding process needs a little bit more CPU time before it voluntarily relinquishes the CPU. It also adjusts PFS to changes based on the feedback obtained from each execution. Also, our experimental results show that processing time can be reduced by using our scheduler.

However, the price to be paid for reducing the processing time is that we reduce the fairness of the system. Even though this work is motivated by the question why can't often-used programs run faster, we need to ensure that the resulting unfairness does not outweigh the performance gains obtained. This could be done by using the early process switching mechanism to compensate for the delay process switching during other times. Also, the impact of varying the maximum dispatch delay time on fairness needs to be explored.

Beside those areas mentioned above, some of our future work will also include evaluating the effectiveness of our scheduler with other programs, exploring the cost of storing PFS on the disk, developing a method to determine which processes running behalf of often used programs, and extending our scheduler to deal with programs consisting of multiple processes.

Acknowledgement

We would like to thank James Michael Perry for his assistance in proofreading this paper.

References

[1] H. Chu and K. Nahrstedt, “A soft real time scheduling server in UNIX operating system,” Technical Report UIUCDCS-R-97-1990, Department of Computer Science, University of Illinois at Urbana Champaign, March 1997.

[2] Silberschatz and P. Galvin, *Operating System Concepts (5th ed.)*, John Wiley & Sons, 1997.

[3] C. Mercer, S. Savage, and H. Tokuda, “Processor capacity reserves for multimedia operating systems,” Technical Report CMU-CS-93-157, School of Computer Science, Carnegie Mellon University, May 1993.

[4] C. Waldspurger and W. Weihl, “Lottery scheduling: flexible scheduling proportional-share resource management,” In *Proc. of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pp.1-11, Nov. 1994.

[5] C. Waldspurger and W. Weihl, “Stride scheduling: deterministic proportional-share resource

management," Technical Report MIT/LCS/TM-528, MIT laboratory for computers science, June 1995.

[6] M. Jones, D. Rosu, and M. Rosu, "CPU reservations and time constraints: efficient, predictable scheduling of independent activities," In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pp.198-211, Oct. 1997.

[7] D. Golub, "Operating system support for coexistence of real-time and conventional scheduling," Technical Report CMU-CS-94-212, School of Computer Science, Carnegie Mellon University, Nov. 1994.

[8] B. Ford and S. Susarla, "CPU inheritance scheduling," In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pp.91-106, Oct. 1996.

[9] P. Goyal, X. Guo, and H. Vin, "A hierarchical CPU scheduler for multimedia operating systems," In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pp.107-121, Oct. 1996.

[10] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel and Dist. Systems*, vol.7, no.5, pp.506-521, May 1996.

[11] H. Wang, A. Nicolau, and K. Siu, "The strict time lower bound and optimal schedules for parallel prefix with resource constraints," *IEEE Trans. Comput.*, vol.45, no.11, pp.1257-1271, Nov. 1996.

[12] M. Wu and W. Shu, "On parallelization of static scheduling algorithms," *IEEE Trans. Software Eng.*, vol.23, no.8, pp.517-528, Aug. 1997.

[13] http://www.microsoft.com/Windows98/guide/Win98/Features/Faster.asp

[14] http://www.microsoft.com/Windows98/usingwindows/maintaining/articles/811Nov/MNTfoundation2a.asp