IEICE TRANS. INF. & SYST., VOL.E83–D, NO.9 SEPTEMBER 2000

PAPER

Evaluation of a Process Scheduling Policy for a WWW Server Based on Its Contents

Sukanya SURANAUWARAT†, *Nonmember and* Hideo TANIGUCHI†, *Regular Member*

SUMMARY Traditional process schedulers in operating systems control the sharing of the processor resources among processes using a fixed scheduling policy based on the utilization of a computer system such as a real-time or a timesharing system. Since the control over processor allocation is based on a fixed policy, not based on *contents* or *behavior of processes*, this can hinder an effective use of a processor or can extend the processing time of a process unnecessarily in some cases. We have already proposed a process scheduling policy, which responds to the behavior of multiple processes of a WWW server, in order to improve the response time of a WWW server. This policy gives any process of a WWW server that is predicted to be a WWW server process handling a text data request from a browser priority over all other processes by moving it to the head of the ready queue where processes waiting for the processor to become available are placed. In this paper, we present the experimental evaluation of our proposed process scheduling policy with regard to the number of simultaneous accesses from browsers and the processor load of the server machine, and explain the results we obtained.

key words: *process scheduling, WWW server, response time, execution behavior, predict*

1. Introduction

Traditional process schedulers in operating systems control the sharing of the processor resources among processes using a fixed scheduling policy based on the utilization of a computer system such as a real-time or a timesharing system. A real-time system's scheduling policy must be able to analyze or handle data faster than they come in and it must also respond to time events. In general, real-time systems are categorized as *hard real-time* and *soft real-time*. In hard real-time systems meeting application-specific deadlines is required, while in soft real-time systems missing a deadline is unfortunate but not catastrophic. Therefore, scheduling applications in hard real-time systems has been an important area of research in real-time systems (e.g., [1]–[8]). A timesharing system's scheduling policy is to provide good response to interactive users. It is an historical artifact from a time when many users with interactive and batch computing requirements shared a single processor, and it is still used in most workstation operating systems.

Real-time systems' scheduling policies are usually only available in real-time operating systems, and not in more general purpose operating systems, like workstation operating systems. However, the advent of multimedia applications (the applications with real-time characteristics) on PCs and workstations has called for new scheduling paradigms to support real-time in systems with traditional timesharing schedulers. One approach to do this is to schedule based on proportion and/or period [9]–[12]. A different 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 [13]–[15].

Since the traditional process scheduling policies are based on the utilization of a computer system, none of the above approaches is trying to schedule based on *contents* or *behavior of processes*. As a consequence, this can hinder an effective use of a processor or can extend the processing time of a process unnecessarily in some cases. For example, in a timesharing system, if a process does not complete its job before its quantum (time-slice) expires, the processor is preempted and given to the next waiting process no matter how little more processor time the process needs. Thus, a process that needs just a little bit more processor time will not complete its job until its next quantum. Because of this, the processing time and the context switching cost of the process increase unnecessarily. If we had predicted the behavior of the process and delayed process switching based on the predicted behavior that the process needed a little bit more processor time to complete its job, then the extra costs mentioned above would have been avoided.

In a parallel computer system, scheduling a parallel program onto processors is basically done 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 processors of a target machine in a manner which reduces the completion time [16]–[18]. However, scheduling is performed without regard to the previous execution behavior of the parallel program.

Therefore, we proposed the idea called POS (Program Oriented Schedule) [19]. The idea of POS is by increasing operating system ability to alter the execution behavior of a program according to the predicted behavior from the previous execution(s), the operating system could optimize the execution behavior of

Manuscript received October 27, 1999.
Manuscript revised March 23, 2000.
†The authors are with the Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka-shi, 812-8581 Japan.

the program allowing user requirements (e.g., performance enhancement) to be satisfied without making any changes to the existing program. In order to predict the execution behavior of a program, the idea of POS requires that the operating system have the ability to log the execution behavior of the program in terms of process identifier, process state and time. And based on this log, the operating system will create the predicted execution behavior of the program or update it if it already exists.

One function in Windows 98, which users can get "faster program start up" as performance enhancement [20], uses an idea similar to that of POS. That is the function improves the performance of a user's programs based on the previous usage without making any changes to the programs. In other words, 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 [21]. However, the function does not alter the execution behavior of programs based on the previous usage which is what our idea does. So, 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 open a user's programs faster), but it cannot execute or run a user's programs more efficiently.

We have already applied POS to the process scheduler [19], [22]. In [19], we proposed the process scheduling policy that allows a process to continue its execution even though its quantum has already expired when it is predicted that a process needs a little bit more processor time before it issues an I/O operation. The objective of this policy is to minimize processing time and/or context switching cost of the process of the target program. However, the target programs of this policy are the programs that consist of only a single process. So, we extended our work to the programs composed of multiple processes such as servers. Server performance is crucial to client/server applications [23]. We used a WWW server, which is a program that consists of multiple processes, as a sample server. And we proposed a process scheduling policy for improving the response time of a WWW server [22]. This policy gives any process of a WWW server that is predicted to be a WWW server process handling a text data request from a browser priority over all other processes by moving it to the head of the ready queue where processes waiting for the processor to become available are placed. We also evaluated this policy experimentally in simple cases for the purpose of verifying that our scheduling mechanism works as expected.

In this paper, we present the experimental evaluation of the proposed process scheduling policy in more complicated and useful cases compared with those in [22]. First, we measure the performance of a WWW server in terms of response time and compare the performance of the proposed policy to that of a conventional priority-based timesharing policy when the WWW server is accessed by a lot of browsers simultaneously. For a WWW server, this kind of situation is more likely to be a realistic case compared with those in [22] where the number of browsers accessing the WWW server ranged from 1 to 3 and just two machines (a client machine and a server machine) were used. And this could be the situation in which it is most desirable to improve the response time of a WWW server. Second, we measure the effect of the number of simultaneous accesses on the processing ability of the server machine in terms of response time and processor load. Then, we find the relation between the number of simultaneous accesses and the response time, and the relation between the number of simultaneous accesses and the processor load. Based on the number of simultaneous accesses, we also compare the response time of the proposed policy to that of a conventional priority-based timesharing policy.

The rest of this paper is organized as follows. Section 2 gives an easy example to show how the performance of a program will be improved when POS is applied to the process scheduler. Section 3 briefly overviews our scheduling mechanism. Section 4 describes our experiments and explains the results we obtained. Section 5 offers our conclusions and future work.

2. An Example of the Effect of POS

We used an easy example shown in Fig. 1 to identify how performance will be improved when POS is applied to the process scheduler.

In Fig. 1, process A and process B need respectively 3.4 seconds and 2.1 seconds of processor time to accomplish their jobs. Both processes have the same priority and a time-slice of 1 second. Figure 1 (a) shows the processing times of process A and process B when using a conventional priority-based timesharing scheduler.

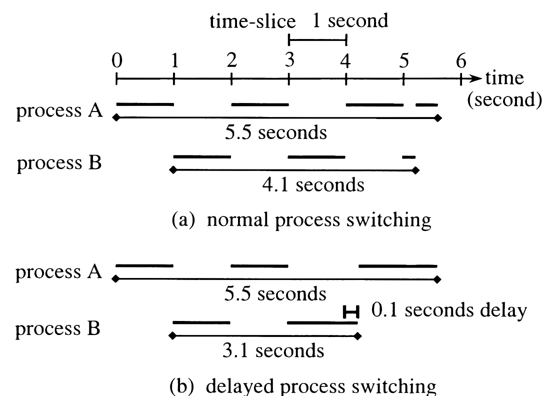


Fig. 1 An example of the effect of POS.

The processing times of process A and process B are 5.5 seconds and 4.1 seconds respectively. On the other hand, Fig. 1 (b) shows the processing times of process A and process B when POS is applied. Based on the predicted behavior that process B needs only 0.1 seconds more processor time to complete its job, the process 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 Fig. 1 (a). Moreover, the context switching cost of process A and B also decreases.

This example also shows how the process scheduling policy, we proposed in [19], controls the time-slice length of the object process, and how the processing time and/or context switching cost of the object process will decrease when using this policy.

3. Overview

Our process scheduling mechanism is composed of two parts: the *logging mechanism* and the *process control mechanism*. We log the execution behavior of a WWW server and create/update the predicted execution behavior called PFS (Program Flow Sequence) through the logging mechanism. And we alter the execution behavior of the WWW server by using the process control mechanism.

3.1 Logging Mechanism

We will first briefly sketch how the WWW server we used forks and maintains each process of the WWW server to provide sufficient background information, and then proceed to the description of the logging mechanism.

The WWW server software we used is Apache version 1.2.5. Apache can be run in two different modes: from the *inetd* system process or, in standalone mode. Standalone is the most common mode of operation, since it is far more efficient. Therefore, our Apache server was configured to run in standalone mode, and how it operates is described as following. On startup, the parent server process creates a pre-defined number of child server processes. From this point on, the parent server process checks the status of the child server processes periodically. If the number of *idle* child server processes falls below the pre-defined lower bound, extra child server processes are created, one per second. An idle child server process is one which not handling a request. If the number of idle child server processes exceeds the pre-defined upper bound, the extra child server processes are killed off. Beside this, there are also pre-defined upper bounds for the number of requests each child is allowed to process before it dies, and on the number of simultaneous requests that can be served; not more than this number of child server processes will be created. Note that our scheduling mechanism is interested in only the child server process(es) and refers to each one as a WWW server process.

When the WWW server is running, a log is collected. A log is a sequence of entries describing process identifier, process state and time. And based on this log a sequence called PFS, which is the predicted behavior of a process, is created or updated for each WWW server process. PFS is a sequence of entries describing process state and time spent.

3.2 Process Control Mechanism

For this paper, the content of a WWW page is pretty simple, that is, one which contains just text data and image data. Text data and image data are separately saved in a file written in HTML (HTML file) and an image-formatted file (Image file) respectively. After making a request for a WWW page, the browser will interpret and process the HTML file sent back by the WWW server it requested and then display the text data. During the interpretation, if the WWW page also contains image data, then the browser will request the WWW server again for the Image file.

From the point of view of WWW users, they want fast response time (i.e., the time from requesting a WWW page until text data displays). Since the rise of the WWW, we have seen a commensurate rise in *impatience*. Nowadays, people including us get frustrated if it takes a long time to download a page. Therefore, improving the response time for the users we serve is important, and improving the performance of WWW servers is vital to the goal of reducing response times for WWW users.

When a WWW server is accessed by a lot of browsers simultaneously, it takes time even for the text data which is much smaller than image data to show up on browsers. As a result, the response time of each user decreases significantly. This situation could be one in which it is most desirable to improve the response time of a WWW server. Hence, in such a situation, our scheduling goal is to give the user something to read (as soon as possible) while the images are coming in and also to allow the user to stop loading if the page is not sufficiently interesting to warrant waiting. A method to achieve this goal is described below. Note that since POS idea-based scheduling is based on the behavior of processes of programs, the delayed process switching method mentioned in Sect. 2 cannot be applied to the WWW server. Because this method is just an example of the effect of POS; so the behavior of processes of programs it assumes are also simple and totally different from the behavior of WWW server processes.

Most processes, except the currently executing process (i.e., process that is in the run state), are in one of two queues: a ready queue or a sleep queue. Processes that are waiting for the processor to become

available (i.e., in the ready state) are placed on a ready queue, whereas processes that are blocked awaiting an event (i.e., in the wait state) are located on a sleep queue associated with the event. When a process is blocked awaiting an event to happen, if the resources (e.g., a hard disk) needed for the event are being used by any other process, then that process needs to wait first for those resources to become available. Next, that process needs to wait again for the operation (e.g., input/output) it initiated to be completed. By reducing the time waiting for the processor to become available in the ready queue or for the resource needed for an event to become available in the sleep queue, we can achieve an enhanced response time. According to this, we proposed the process scheduling policy that when a processor (a hard disk or a network communication) becomes bottlenecked, any WWW server process handling an HTML file will be moved to the head of the ready queue (sleep queue associated with the event) [22]. Note that the bottleneck of the processor mentioned in this paper is the situation in which the processor is busy and there is more than one process waiting in the ready queue.

When we discussed the process control mechanism that implements the above policy focused on the bottleneck of the processor, we had two problems: how to detect which processes are WWW server processes handling HTML files, and how to operate the ready queue.

To answer these questions we found that we needed to look at the detailed execution behavior of a WWW server. We analyzed the behavior of WWW server processes based on PFS and found out that any WWW server process handling an HTML file has *2* characteristics: *it runs after waiting for a long time in the wait state* (characteristic 1) and *it tends to cycle between run state and wait state a number of times but fewer times than that of WWW server process handling an Image file* (characteristic 2).

To deal with the fist problem, we introduced two parameters into our process control mechanism in order to determine which processes are WWW server processes handling HTML files: long wait threshold (its value is denoted by SLP) and run state/wait state threshold (its value is denoted by RW). If the time spent by a process in the wait state before moving to the run state is more than SLP, and the number of times the process changes between run state and wait state is less than RW, then we determine that it is a WWW server process handling an HTML file. By these two parameters, we can detect which process appears to be a WWW server process handling an HTML file.

To deal with the second problem, our process control mechanism puts any process that has characteristic 1 at the head of ready queue and moves that process to the back of the ready queue when that process loses characteristic 2. The reason processes that lose characteristic 2 are moved to the back of the ready queue is that sometimes WWW server processes handling Image files are mistaken as WWW server processes handling HTML files because they exhibit characteristic 1.

How to predict and update SLP: We analyzed the execution behavior of a WWW server based on PFS and found that a WWW server process handling an HTML file is the process that waits for a request from a browser. The time it waits for a request is relatively long. Therefore, the longest time of each WWW server process in the wait state is determined from PFS for each period, then SLP for the next time period is set to the smallest of these values. SLP is updated every time period.

How to predict and update RW: We analyzed the execution behavior of a WWW server based on PFS and found that the number of times a WWW server process changes between run state and wait state is proportional to the size of the files they handle (i.e., HTML file or Image file). Since HTML files are generally smaller than Image files, the number of times a WWW server process handling an HTML file changes between run state and wait state is smaller than that of a WWW server process handling an Image file. Therefore, the smallest number of times each WWW server process changes between run state and wait state is determined from PFS for each period, then RW for the next time period is set to the greatest of these values. RW is updated every time period.

We also note that it can be thought that our scheduling policy is similar to the preemptive shortest job first policy (i.e., the waiting process with the smallest estimated processing time is run next) due to the operation of moving WWW server processes handling HTML files, whose processing time is also proportion to the size of the files they handle, to the head of the ready queue. However, the point of POS idea-based scheduling policies is that the decision of which process to run next is performed based on the behavior of processes which is not limited to which process has the shortest processing time.

4. Measures of Performance

In this section, we present experiments designed to evaluate the effectiveness of our scheduling mechanism. We start with a description of the experimental setup, and proceed to present the results of experiments.

4.1 Experimental Setup

Our scheduling mechanism is implemented in BSD/OS version 2.1. The software used for the WWW server and the browser in our experiment was Apache version 1.2.5 and Netscape Navigator version 3.04 respectively. The WWW server ran on a personal computer with a 233 MHz AMD-K6 processor and 64 MB of memory,

while browsers ran on three personal computers, each with a 200 MHz Intel Pentium Pro processor and 64 MB of memory. All machines were running on BSD/OS version 2.1 and were connected by a private 10 Mb/s Ethernet. All our experiments were conducted in single user mode, and the operating system's I/O buffer cache in the server machine and each browser's cache were disabled during the experiments in order to see the effect of our scheduling mechanism clearly.

The WWW server was accessed by three browsers from each of the three machines simultaneously. All browsers accessed unique URLs all of which have the same content. In three different experiments in which we varied RW in the range from 1 to 10, we measured the time (t1) from requesting a WWW page until text data starts displaying and the time (t2) from requesting a WWW page until image data displays completely for each access, and then found the average of the 5 trial times of t1 (response time of text data) and t2 (response time of image data). In experiment 1, all the browsers accessed the WWW server simultaneously every 30 seconds when the WWW server coexisted with a processor-bound process and SLP was fixed at 20 seconds. The purpose of this experiment is to know how the response time of text data would be improved in the situation that is the best for our scheduling mechanism (i.e., the processor of the server machine is bottlenecked [caused by a coexisting processor-bound process] and accesses from browsers are set in such a way that SLP will be predicted 100% correctly). In experiment 2, all the browsers accessed the WWW server randomly at the same time and SLP was fixed at 20 seconds, while in experiment 3, SLP was predicted and updated automatically based on PFS every 500 milliseconds. There was no processor-bound process in experiments 2 and 3. The purpose of experiments 2 and 3 is to know how in a more realistic situation (i.e., no processor-bound process coexists and accesses from browsers are random) the response time of text data would be improved when SLP was fixed and when SLP dynamically varies according to the execution behavior of the WWW server.

4.2 Experimental Results

Figure 2 shows some examples of the results of experiment 1, when not using and using our scheduling mechanism (RW = 3,6,9). We used a conventional priority-based timesharing mechanism when not using ours. Figure 2 plots the URLs in numerical sequence on the y-axis against the response time of text data to a request in seconds. Figure 2 shows that the minimum and the maximum response times when RW = 3,6,9 are better than when not using our scheduling mechanism. This implies that the range or the variance of response times becomes narrower when using our scheduling mechanism. In addition, Fig. 2 also shows that by using our scheduling mechanism, the extreme values of response times such as the case when URL = 1,10,11 are pressed down.

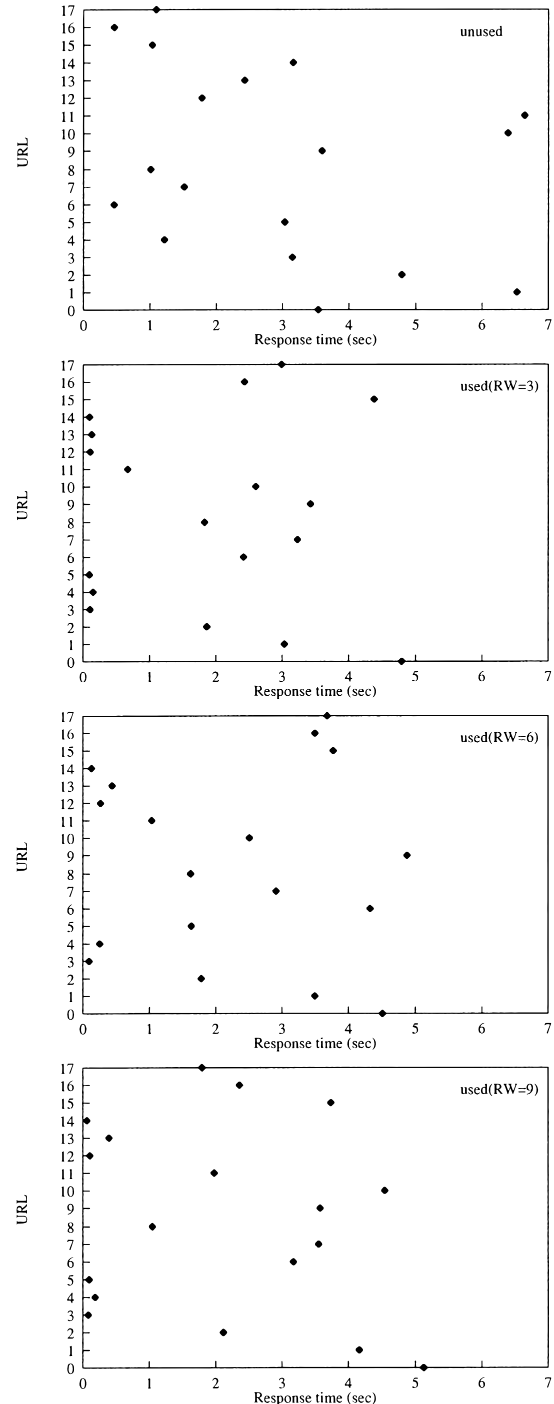
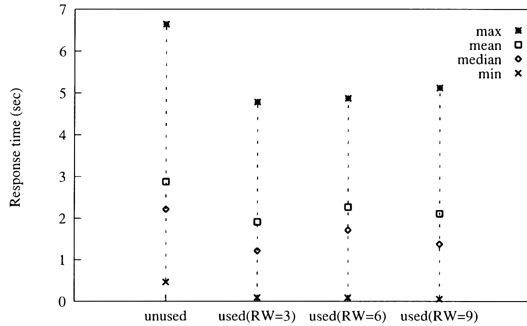
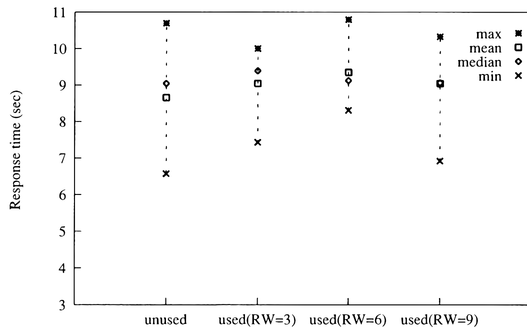


Fig. 2 Examples of the response time of text data in experiment 1.

In order to make the experimental results easier to understand and discuss, we put all the data shown in Fig. 2 into one graph as shown in Fig. 3 (a). Figure 3 (a) illustrates the minimum, the maximum, the mean, the median and the range of response times of

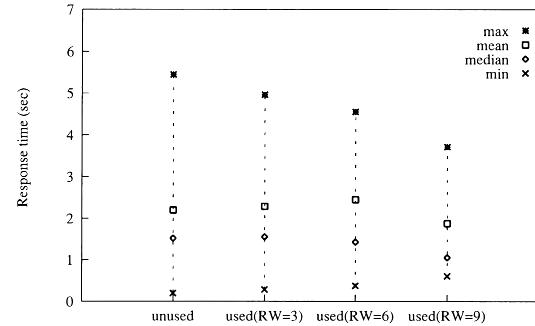


(a) Response time of text data

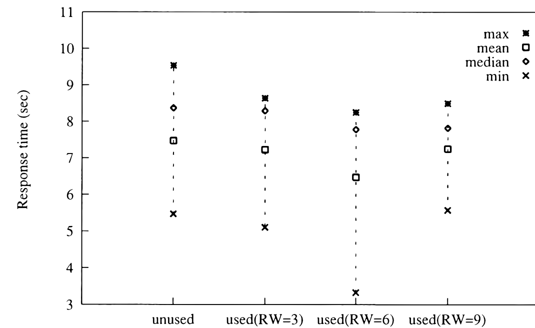


(b) Response time of image data

Fig. 3 The effect of our scheduling mechanism in experiment 1.



(a) Response time of text data



(b) Response time of image data

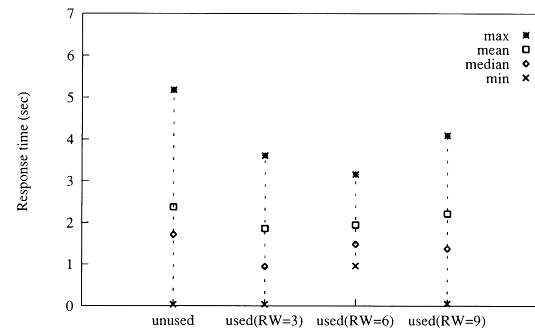
Fig. 4 The effect of our scheduling mechanism in experiment 2.

text data to a request in seconds. Note that the median values are calculated directly from t1, since it can be skewed if we calculate it from the response times shown in Fig. 2, which are the average of the 5 trial times of t1. This figure shows that the mean response times of text data when RW = 3,6,9 are improved 33.8%, 21.3% and 26.6% respectively, while the median response times are improved 44.7%, 22.8% and 37.7% respectively. These figures are calculated by $\frac{a-b}{a} \times 100\%$ where a and b are the mean or the median response times when not using and using our scheduling mechanism respectively. This case produced the most improvement of the response time of text data, because the coexisting processor-bound process always caused the processor to become bottlenecked and the WWW server processes waiting for HTML file requests from browsers were always in the wait state at least 30 seconds which was more than SLP (20 seconds).

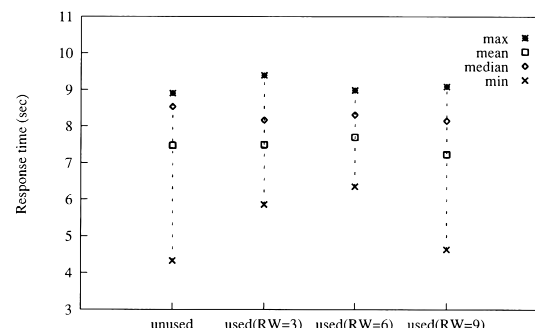
The rest of the experimental results will be shown like Fig. 3 (a).

Figure 3 (b) illustrates the minimum, the maximum, the mean, the median, and the range of response times of image data to a request in seconds. This figure shows that the minimum, the mean and the median response times when RW = 3,6,9 are not as fast as when not using our scheduling mechanism, even though they are close in some cases. This is expected and is due to our policy of giving processes handling HTML files priority over all other processes including WWW server processes handling Image files.

Figures 4 and 5 show respectively the results of experiments 2 and 3 when not using and using our scheduling mechanism (RW = 3,6,9). Figures 4 (a) and 5 (a) illustrate the minimum, the maximum, the mean, the median and the range of response times of text data to a request in seconds. In Fig. 4 (a), we did not notice



(a) Response time of text data



(b) Response time of image data

Fig. 5 The effect of our scheduling mechanism in experiment 3.

any consistent improvement when using our scheduling mechanism, even though the mean and the median response time of text data when RW = 9 is faster than when not using it. On the other hand, in Fig. 5 (a), although the minimum response times for RW = 6 are not better than when not using our scheduling mechanism, the extreme values of response times are pressed down. The mean response times of text data when RW = 3,6,9 are improved 22.2%, 17.9% and 6.7% respectively, while the median response times are improved 44.6%, 13.0% and 19.3% respectively.

Even though a processor-bound process causing the bottleneck of the server machine does not coexist in Fig. 5 (a), our scheduling mechanism still produces a good improvement. This might imply that a lot of simultaneous accesses from browsers could cause the processor of the server machine to become bottlenecked. From now on, our explanations about experiments 2 and 3 will be done on the supposition that a lot of simultaneous accesses from browsers cause the processor of the server machine to become bottlenecked.

The only difference between experiments 2 and 3 is the way SLP was set. In experiment 2, the results of response times of text data shown in Fig. 4 (a) indicate that the WWW server processes waiting for HTML file requests from browsers might not be in the wait state more than SLP, because SLP was fixed at 20 seconds while browsers accessed the WWW server randomly. On the other hand, the improvement of response times of text data in experiment 3 shown in Fig. 5 (a) means that SLP is accurately predicted and updated by our scheduling mechanism based on the execution behavior of the WWW server, because SLP is predicted and updated automatically based on PFS every 500 milliseconds.

Figures 4 (b) and 5 (b) illustrate the minimum, the maximum, the mean, the median, and the range of response times of image data to a request in seconds respectively. In Fig. 4 (b), the mean and the median response times of image data are improved. This could be because sometimes WWW server processes handling Image files were mistaken as WWW server processes handling HTML files when they were in the wait state more than SLP (20 seconds). In Fig. 5 (b), the mean response times of image data when RW = 3,6,9 are not so different from when not using our scheduling mechanism, while the median response times are smaller; which means the number of small response times are increased when using our scheduling mechanism.

When WWW server process handling Image files are in the wait state more than SLP (i.e., when they are in the wait state for a long time), our scheduling mechanism will distinguish them from WWW server processes handling HTML files, by comparing the number of times they change between run state and wait state with RW. These WWW server processes handling Image files will be treat as WWW server processes handling HTML files until the number of times they change between run state and wait state is more than RW. Therefore, the closer RW is set to the number of times WWW server process handling HTML files actually change between run state and wait state, the less benefit the WWW server processes handling Image files in the case mentioned above will take from being treated as WWW server processes handling HTML files. This requires that RW also be predicted and updated automatically based on PFS, which is part of our future work. We examined the PFS and found out that the number of times a WWW server process handling an HTML file changes between run state and wait state was about three or four. That's why, in Fig. 3 (a) and Fig. 5 (a), our scheduling mechanism produces the best improvement when RW = 3 and the improvement declined when RW was higher.

Summarize experimental results pertaining to response time of text data: The results of experiment 1 show that the mean and the median response times of text data are improved greatly (up to 33.8% and 44.7% respectively when RW = 3) when the processor of the server machine is bottlenecked which is the condition that our mechanism favors and SLP is predicted 100% correctly. The results of experiment 2 shows that the mean and the median response times of text data are not improved at all when SLP is fixed, while those of text data in experiment 3 are improved (up to 22.2% and 44.6% respectively when RW = 3) when SLP is predicted and updated based on PFS. In experiment 3, even though a processor-bound process causing the bottleneck of the server machine does not coexist, our scheduling mechanism still produces a good improvement. This might imply that a lot of simultaneous accesses from browsers could cause the processor of the server machine to become bottlenecked. If this supposition is true, then the improvement in experiment 3 means that SLP is accurately predicted and updated by our scheduling mechanism based on the execution behavior of the WWW server, because the only difference between experiments 2 and 3 is the way SLP was set.

4.3 Processor Load

In order to prove our supposition about experiment 3, we decided to find the relation between the number of simultaneous accesses and the response time, and the relation between the number of simultaneous accesses and the processor load (PL) or the length of ready queue (LRQ) by setting up experiment 4. PL shows if the processor is in use or not and LRQ indicates the number of processes which are ready to run but are waiting in the ready queue for the processor to become available. If the processor is in use and the length of the ready queue is not zero, then there is some contention for the processor and a bottleneck exists.

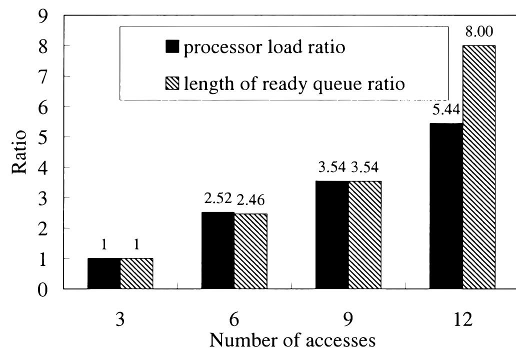


Fig. 6 The relation between the number of simultaneous accesses and the processor load or the length of ready queue.

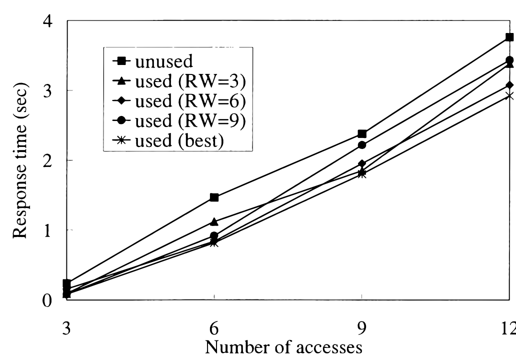


Fig. 7 The relation between the number of simultaneous accesses and the response time.

In experiment 4, we varied the number of client machines from 1 to 4 in which each machine has the same specification described in Sect. 4.1 and runs three browsers, we measured the response times of text data, PL and LRQ of the server machine under the same conditions as in experiment 3. PL is measured by checking if the processor is busy (i.e., in use) or idle every 10 milliseconds. If the processor is busy, PL is incremented. The PL ratio is the ratio of PL when the number of client machines is 1,2,3 and 4 to PL when the number of client machines is 1. LRQ is determined by the summation of the total number of processes in the ready queue every 10 milliseconds. The LRQ ratio is the ratio of LRQ when the number of machines is 1,2,3 and 4 to LRQ when the number of machines is 1.

The results of experiment 4 are shown in Figs. 6 and 7. Figure 6 plots the PL ratio (1:2.52:3.54:5.44) and the LRQ ratio (1:2.46:3.54:8). These ratios show that PL and LRQ increased at almost the same rate except when the number of simultaneous accesses is 12. Figure 6 shows that the more simultaneous accesses there are, the more PL and LRQ will increase. Therefore, a lot of simultaneous accesses cause the processor to become busy or bottlenecked and also increase LRQ. Figure 7 shows the mean response times of text data while not using our scheduling mechanism, using our scheduling mechanism for RW = 3,6,9 and when our mechanism produced the best result. Figure 7 shows that the more simultaneous accesses there are, the worse response time will be. Due to our policy of moving any server process handling an HTML file to the head of the ready queue when the processor becomes bottlenecked, which in this experiment is caused by the simultaneous accesses as indicated by the PL and the LRQ ratios in Fig. 6, the mean response times when using our scheduling mechanism shown in Fig. 7 are always better than when not using it. And this is the reason for the improvement in experiment 3.

In addition, in Fig. 7, the best mean response times, when the number of simultaneous accesses is 3,6,9 and 12, are improved 65.7%, 44.4%, 24.2% and 22.3% respectively. Although the percentage of improvement declined as the increase in the number of simultaneous accesses caused PL to increase, the real time improvement was greater. In addition, the effect of PL on the improvement dropped significantly when the number of simultaneous accesses was 12, by that point LRQ was increasing at a higher rate than PL. And this could be the result of our scheduling policy's manipulation on the ready queue.

Over time, the WWW server in these experiments was not busy. However, when thinking about the short period of time that the WWW server was accessed by a number of browsers at the same time, it was busy which is indicated by the PL and the LRQ ratios shown in Fig. 6. According to the experimental results shown in Fig. 7, any WWW server which experiences a lot of simultaneous accesses from browsers would benefit from our scheduling mechanism.

5. Conclusions

Since the control over processor allocation is based on a fixed policy which is determined by the utilization of a computer system, not based on contents or behavior of processes, this can hinder an effective use of a processor or can extend the processing time of a process unnecessarily in some cases. Therefore, we proposed the idea called POS (Program Oriented Schedule). The idea of POS is by increasing operating system ability to alter the execution behavior of a program according to the predicted behavior from the previous execution(s), the operating system could optimize the execution behavior of the program allowing user requirements, such as performance enhancement, to be satisfied without making any changes to the existing program. We have already applied this idea to the process scheduler and proposed a process scheduling policy for improving the response time of a WWW server. We have also evaluated this policy experimentally in simple cases for the purpose of verifying that our scheduling mechanism works as expected.

In this paper, we evaluated the effectiveness of our proposed process scheduling policy by measuring the

response time of a WWW server when it is accessed by a lot of browsers simultaneously, which could be the situation in which it is most desirable to improve the response time of a WWW server. Our experimental results show that the mean response times are improved greatly up to 33.8% in the best case in which the processor of the server machine is bottlenecked caused by a coexisting processor-bound process and accesses from browsers are set in such a way that the scheduling parameter SLP will be predicted 100% correctly. In a more realistic case in which no processor-bound process coexists and accesses from browsers are random, our scheduling mechanism still produces a good improvement of up to 22.2% when the scheduling parameter SLP is predicted and updated automatically by our scheduling mechanism based on the predicted execution behavior of the WWW server, called PFS. PFS is created and updated by our scheduling mechanism from the previous execution(s) of the WWW server. Due to the fact that a lot of simultaneous accesses cause the processor of the server machine to become bottlenecked and increase the length of ready queue, this improvement indicates that the scheduling parameter SLP is accurately predicted and updated by our scheduling mechanism based on the execution behavior of the WWW server. Therefore, any WWW server which experiences a lot of simultaneous accesses from browsers would benefit from our scheduling mechanism.

Future work is required to measure the performance of a WWW server when the scheduling parameter RW is automatically predicted and updated by our scheduling mechanism and also when both scheduling parameter SLP and RW are automatically predicted and updated by our scheduling mechanism.

Acknowledgement

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

References

[1] J. Xu and D.L. Parnas, "On satisfying timing constraints in hard-real-time systems," IEEE Trans. Software Eng., vol.19, no.1, pp.70–84, Jan. 1993.

[2] K. Ramamritham, J.A. Stankovic, and P. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," IEEE Trans. Parallel & Dist. Systems, vol.1, no.2, pp.184–194, April 1990.

[3] T. Shepard and J.A.M. Gagné, "A pre-run-time scheduling algorithm for hard real-time systems," IEEE Trans. Software Eng., vol.17, no.7, pp.669–677, July 1991.

[4] K. Schwan and H. Zhou, "Dynamic scheduling of hard real-time tasks and real-time threads," IEEE Trans. Software Eng., vol.18, no.8, pp.736–748, Aug. 1992.

[5] W.K. Shih, J.W.S. Liu, and C.L. Liu, "Modified rate-monotonic algorithm for scheduling periodic jobs with deferred deadlines," IEEE Trans. Software Eng., vol.19, no.12, pp.1171–1179, Dec. 1993.

[6] M.G. Härbour, M.H. Klein, and J.P. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," IEEE Trans. Software Eng., vol.20, no.1, pp.13–28, Jan. 1994.

[7] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," IEEE Trans. Software Eng., vol.21, no.5, pp.475–479, May 1995.

[8] W. Feng and J.W.S. Liu, "Algorithms for scheduling real-time tasks with input error and end-to-end deadlines," IEEE Trans. Software Eng., vol.23, no.2, pp.93–106, Feb. 1997.

[9] C.W. 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.

[10] C.A. Waldspurger and W.E. Weihl, "Lottery scheduling: Flexible scheduling proportional-share resource management," Proc. 1st USENIX Symposium on Operating Systems Design and Implementation, pp.1–11, Nov. 1994.

[11] C.A. Waldspurger and W.E. Weihl, "Stride scheduling: Deterministic proportional-share resource management," Technical Report MIT/LCS/TM-528, MIT Laboratory for Computers Science, June 1995.

[12] M.B. Jones, D. Rosu, and M.-C. Rosu, "CPU reservations and time constraints: Efficient, predictable scheduling of independent activities," Proc. 16th ACM Symposium on Operating Systems Principles, pp.198–211, Oct. 1997.

[13] D.B. 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.

[14] B. Ford and S. Susarla, "CPU inheritance scheduling," Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation, pp.91–106, Oct. 1996.

[15] P. Goyal, X. Guo, and H.M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation, pp.107–121, Oct. 1996.

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

[17] H. Wang, A. Nicolau, and K.S. 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.

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

[19] H. Taniguchi, "POS: Program oriented schedule," IPS Japan Proc. Computer System Symposium'96, vol.96, no.7, pp.123–130, 1996.

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

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

[22] S. Suranauwarat and H. Taniguchi, "Process scheduling policy for a WWW server based on its contents," Trans. IPS Japan, vol.40, no.6, pp.2510–2522, June 1999.

[23] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie, "Application performance and flexibility on exokernel systems," Proc. 16th ACM Symposium on Operating Systems Principles, pp.52–65, 1997.

Sukanya Suranauwarat received the B.S. and M.S. degrees in computer science from Kyushu University, Japan, in 1997 and 1999, respectively. Presently, working toward a doctor's degree in the Graduate School of Information Science and Electrical Engineering at Kyushu University. Her research interests include operating systems and distributed processing.

Hideo Taniguchi received the B.E. degree in 1978, the M.E. degree in 1980, and the Ph.D. degree in computer science in 1991, all from the Kyushu University, Fukuoka, Japan. In 1980, he joined NTT Electrical Communication Laboratories. In 1988, he moved Research and Development headquarters, NTT DATA Communications Systems Corporation. He had been an Associate Professor of Computer Science at Kyushu University since 1993. He has been an Associate Professor of Graduate School of Information Science and Electrical Engineering at Kyushu University since 1996. His research interests include operating system and distributed processing. He is a member of the Information Processing Society of Japan and ACM.