OPERATING SYSTEMS SUPPORT FOR SOFTWARE MAINTENANCE: AN ASSESSMENT USING WWW SERVER SOFTWARE

SUKANYA SURANAUWARAT HIDEO TANIGUCHI

Graduate School of Information Science and Electrical Engineering,
Kyushu University, Fukuoka 812-8581, Japan

ABSTRACT

Improving an operating system's support for software maintenance is, we believe, vital to our goal of reducing the significant sum spent on adapting existing software to changing user requirements, especially improving the performance of software. Therefore, we proposed the idea that by increasing an operating system's abilities to observe a software's execution behavior and evolve its execution behavior using observed results, an operating system could adapt existing software to changing user requirements without making any modifications to the software. We have already integrated the above abilities into CPU and disk scheduling mechanisms in an operating system. In this paper, we verify the usefulness of our idea using existing software like a WWW (World Wide Web) server, by examining its performance when using both our CPU and disk scheduling mechanisms.

KEY WORDS: Software Maintenance, WWW Server, Operating System, Behavior, Predict

1. INTRODUCTION

The magnitude of software maintenance cost is estimated to comprise at least 50% of overall software life-cycle cost [1][2][3]. And a large portion of this cost, over 50%, is spent on changes to accommodate changing user requirements. Changes in user requirements are inevitable. Software models part of reality, and reality changes. So the software has to change too. Keep this in mind, our goal is to reduce the significant sum spent on changing user requirements, especially performance enhancement of software. This goal could be achieved by using the following idea.

Our idea is that by increasing an operating system's abilities to *observe* a software's execution behavior and *evolve* its execution behavior using observed results, an operating system could *adapt* existing software to changing user requirements *without making any modifications* to the software. In other words, by using these abilities, an operating system could optimize a software's execution behavior allowing user requirements to be satisfied without any modifications to the existing software.

We have already integrated the above abilities into resource scheduling mechanisms in an operating system such as CPU and disk scheduling mechanisms. We have also verified the usefulness of our idea in some cases using existing software, i.e., a WWW (World Wide Web) server. That is, we evaluated the performance of a WWW server when using either scheduling mechanism by itself. In either case, our mechanism alters the execution behavior of a WWW server by giving preferential use of the resource (i.e., the CPU resource or the disk drive) to any process that is predicted to be a server process handling an HTML (HyperText Markup Language) file request. This allows users to view the first data to display on browsers, the text data stored in an HTML file, and the general layout of a WWW page in a timely manner during periods of high demand. As a result, the user requirement, the enhancement of response time when there is a heavy demand on the server, can be satisfied without making any changes to the existing server software.

In this paper, we present experimental evaluation of our idea by examining the performance of a WWW server when using both our CPU and disk scheduling mechanisms. And our result shows that the mean response time is improved as much as 30%; this percentage of improvement is better than when using either mechanism by itself (both of which are about 20%).

The remainder of this paper is organized as follows. Section 2 provides background information on the WWW. Section 3 is a brief overview of our CPU and disk scheduling mechanisms. Section 4 describes our experiment and explains the results we obtained. Section 6 offers our conclusion and future work.

2. THE WWW

The WWW is based on the client-server model. That is, users access WWW pages provided by WWW servers via WWW clients, mainly browsers. WWW pages are written in HTML and stored on a disk as text files called HTML files. An HTML file contains text data that users will view and HTML tags that specify structure for the text data as well as formatting hints. Because an HTML file uses a text representation, non-text data such as images are not included directly in the HTML file. Instead, a tag is placed in the HTML file to specify the place at which an image should be inserted and the source of the file that stores it (Image file). HTML files and Image files account for more than 90% of the total requests to a server [4][5]. Therefore, the WWW page in this paper consists of an HTML file and an Image file.

WWW clients. A user can access the information on the WWW by using a browser, such as Netscape Navigator, Internet Explorer, or Mosaic. When the user selects a WWW page to retrieve (usually, by clicking a mouse on a hyperlink), the browser creates a request to be sent to the corresponding WWW server and then waits for a response. When the response arrives, the browser interprets and processes the HTML file sent back by the server, and then displays the text data for the user to view. During the interpretation, if the WWW page also contains other types of data such as an image, then the browser will request the Image file from the WWW server.

WWW servers. The purpose of a WWW server is to provide WWW pages to WWW clients that request them. The server software we used is Apache version 1.2.5 [6], the pre-forking model server. In this model, a *master* process pre-forks a pool of *child* server processes to handle requests. However, the master process does not handle any part of the request. In this paper, we refer to each child server process as a server process.

3. OVERVIEW

In this section, we will briefly describe our CPU and disk scheduling policies and their aspects of implementation [7][8] in order to provide sufficient understanding to the rest of the paper.

3.1 SCHEDULING POLICIES

As the demand placed on a WWW server grows, the number of simultaneous requests it must handle increases. As a result, users see slower response times during periods of high demand. In other words, it takes a longer time for the first data to display on browsers, the text data stored in an HTML file, to start displaying when a WWW server is accessed by many browsers simultaneously. This situation could be one in which it is most desirable for users to improve the response time of a WWW server, since they tend to get frustrated if it takes a long time to start viewing a WWW page. Hence, in such a situation, our scheduling goal is to display the text data for the user to view as soon as possible while other types of data such as an image 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.

Most processes, except the currently executing process (i.e., process in the *run* state), are in one of two queues: a ready queue or a sleep queue. Processes that are waiting for the CPU to become available (i.e., processes in the *ready* state) are placed on a ready queue, whereas processes that are blocked awaiting an event (i.e., processes in the *wait* state) are located on a sleep queue associated with the event. When a process is blocked awaiting an event to happen such as the completion of its I/O request, if the desired I/O device (e.g., a disk drive) is available, the request can be serviced immediately. If that device is being used by any other process, then the request will be put into the I/O queue for that device. By reducing the time a process waits for the CPU to become available in the ready queue or the time its I/O requests wait to be serviced in the I/O queue, we can reduce the processing time of a process, which results in an enhanced response time if that process is a server process handling an HTML file request. For this reason, we proposed the following scheduling policies.

(1) When a CPU becomes bottlenecked, any server process handling an HTML file request will be moved to the head of the ready queue.

(2) When an I/O device becomes bottlenecked, any I/O request generated by any server process handling an HTML file request will be moved to the head of the I/O queue for that device.

Note that the bottleneck mentioned in this paper is the situation in which the resources are being used and there is more than one process waiting to use the resources.

3.2 ASPECTS OF IMPLEMENTATION

When implementing the proposed scheduling policies focused on the bottleneck of a CPU and a disk drive, we had *two* problems: *how to detect which processes are server processes handling HTML file requests*, and *how to operate the ready queue and the I/O queue*.

To deal with the first problem, we first need to know what a server process handling an HTML file request is like. So, we logged the execution behavior of a WWW server in terms of process identifier, process state and time. And based on this log, we created the predicted execution behavior called *PFS (Program Flow Sequence)* for each server process. PFS is a sequence of entries describing process state and time spent. We analyzed the behavior of server processes based on PFS and found that a server process handling an HTML file request is a process that waits for a request from a browser, and the time it waits for a request is relatively long compared with the time waiting for other kinds of events to happen (e.g., the completion of an I/O request) in the wait state. Also, after accepting a request, it tends to change between run state and wait state a number of times. The number of changes is proportional to the size of the file it handles, i.e., an HTML file (which is usually smaller than an Image file). In other words, any server process handling an HTML file request has two characteristics: after waiting for a long time in the wait state (characteristic 1), it tends to cycle between run state and wait state a number of times but fewer times than that of a server process handling an Image file request (characteristic 2).

Next, we introduced two parameters into our scheduling mechanisms in order to determine which processes have the above two characteristics (i.e., to determine which processes are server processes handling HTML file requests): 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 server process handling an HTML file request. By these two parameters, we can detect which process appears to be a server process handling an HTML file request. SLP and RW are automatically predicted and updated every time period based on the predicted execution behavior of each server process, i.e., PFS of each server process. We note that we cannot fix SLP and RW at some values due to random accesses from users (in the case of SLP), and the changing execution behavior of the server and the size of the HTML files it handles (in the case of RW). Also, PFS is created and updated every time period based on the log we collected when the WWW server is running. As mentioned, a log is a sequence of entries describing process identifier, process state and time.

To deal with the second problem, our CPU and disk scheduling mechanisms give preferential use of the CPU resource and the disk drive to any process that is predicted based on its behavior to be a server process handling an HTML file request, by moving it and its I/O requests to the head of the waiting queue (i.e., the ready queue or the I/O queue). In other words, our scheduling mechanisms put any process that has characteristic 1 and also its I/O requests at the head of the waiting queue; and when that process loses characteristic 2, it and its I/O requests will be scheduled normally, i.e., that process and its I/O requests will be respectively put into the ready queue and the I/O queue using the routines provided by the operating system, which in our case are the *roundrobin* and the *disksort* routines. Roundrobin enters processes into the ready queue on a round-robin basis while disksort enters I/O requests into the I/O queue in a cyclic, ascending, cylinder order as described below.

An I/O queue is made up of one or two lists of requests ordered by cylinder number. The request at the front of the first list indicates the current position of the drive. If a second list is present, it is made up of requests that lie before the current position. Each new request is sorted into either the first or the second list, according to the request's location. When the heads reach the end of the first list, the drive begins servicing the other list. Figure 1(a) shows an example of an I/O queue with requests for I/O to blocks on cylinders 75, 30 and 120 when the request at cylinder 100 is being serviced. For comparison, Figures 1(b) and (c) show respectively how disksort and our disk scheduling mechanism enter the requests from a server process handling an HTML file request on cylinders 140 and 80, in addition to the requests mentioned in Figure 1(a). Note that the request at the head of the queue is the one that is being serviced. So, we decided to enter the request of any server process handling an HTML file request right after the one at the head. If there is more than one request generated by server processes handling HTML file requests, then they will be put into the queue on a first-come-first-served basis. In the same way, if there is more than one server process handling an HTML file request that is ready to run, then they will be put at the head of the ready queue on a first-come-first-served basis.

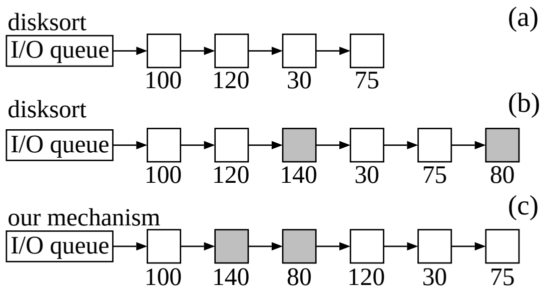


Figure 1. Examples of how disksort and our disk scheduling mechanism enter I/O requests into the queue.

4. EXPERIMENTAL EVALUATION

In this section, we present an experiment designed to examine the performance of a WWW server when using both our CPU and disk scheduling mechanisms. We start with a description of the experimental setup, and proceed to present the results of the experiment.

4.1 EXPERIMENTAL SETUP

Our CPU and disk scheduling mechanisms are 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

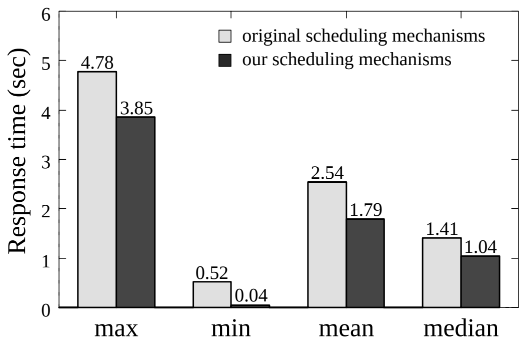
3.04 respectively. The WWW server ran on a personal computer with a 233 MHz AMD-K6 processor and 64 MB of memory, while browsers ran on three personal computers, each with a 200MHz Intel Pentium Pro processor and 64 MB of memory. All machines were running on BSD/OS version 2.1 and were connected by a private 10 Mb/s Ethernet. Also, our experiment was 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 experiment in order to clearly see the effect of our scheduling mechanisms.

In our experiment, the WWW server was accessed by three browsers from each of the three machines randomly at the same time; this number can cause bottleneck of the CPU and the disk drive, which is indicated by the non-zero length of the ready queue and the I/O queue, during the short period of simultaneous accesses [7][8]. Also, all browsers accessed 18 unique URLs (URL — Uniform Resource Locator) all of which have the same content, an HTML file (1,772 bytes) and an Image file (43,770 bytes). For each URL, we measured the 5 trial times of *time1* and *time2*. Time1 is the time from requesting a WWW page until text data starts displaying. Time2 is the time from requesting a WWW page until image data displays completely. We will refer to the averages of 5 trial times of time1 and time2 as *response time of text data* and *response time of image data* respectively. During the experiment, SLP and RW were automatically predicted and updated based on PFS every 500 milliseconds. And our previous works [7][9] have already showed that SLP and RW are effectively predicted and updated by our scheduling mechanisms.

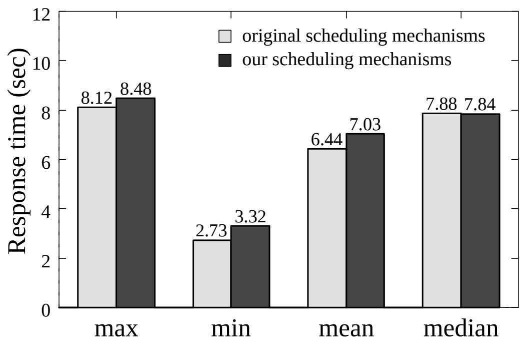
4.2 EXPERIMENTAL RESULTS

Figure 2(a) illustrates the maximum, the minimum, the mean, and the median response times of text data from 18 URLs. Note that the median values are calculated directly from time1, since it can be skewed if we calculate it from the response times of text data, each of which is the average of the 5 trial times of time1. For comparison, we also show the results when using the original scheduling mechanisms (i.e., roundrobin and disksort). Also, the results for image data shown in Figure 2(b) are plotted in the same way.

Figure 2(a) shows that the maximum, the minimum, the mean, and the median response times of text data are improved 20%, 92%, 30% and 26% respectively. These figures are calculated by $\frac{a-b}{a}\times 100\%$ where *a* and *b* are the maximum (or the minimum or the mean or the median) response times of text data when using the original scheduling mechanisms and when using ours respectively. According to the result, our scheduling mechanisms produce a good improvement for response time of text data. In other words, by using our scheduling mechanisms the time from requesting a WWW page until text data starts displaying is reduced. Besides, the mean response time when using both of our scheduling mechanisms simultaneously is improved (30%) more than when using either mechanism by itself (both of which are about 20%). However, the percentage of improvement when using both our CPU and disk scheduling mechanisms at the same time is not the sum of the percentage when using either mechanism by itself. This could be because of the correlation between our two scheduling mechanisms. That is, the operation of moving a process or an I/O request on one queue can affect the operation on the other queue. For example, if a process which is moved to the head of the ready queue by our CPU scheduling mechanism generates an I/O request, then the likelihood that its I/O request will be consequently put into the front of the I/O queue is high. As a result, the effect of moving I/O requests to the head of the I/O queue by our disk scheduling mechanism will be diminished.



(a) Response time of text data



(b) Response time of image data

Figure 2. The experimental results when the WWW server is accessed by multiple browsers randomly at the same time while SLP and RW are predicted/updated automatically.

However, the price to be paid for reducing the time from requesting a WWW page until text data starts displaying is that we reduce the fairness of the system, which consequently affects the amount of time it takes from requesting a WWW page until image data displays completely. If this effect is small, it is acceptable. For example, it is acceptable if the image data displays completely while the users are reading the text data that displays faster. On the other hand, if the effect on the response time of image data is big, then users might get frustrated and not wait for the whole page to display completely. Therefore, care must be taken to ensure that the resulting unfairness does not outweigh the performance gains obtained. And our result in Figure 2(b) shows that the worst or the maximum response time of image data when using our mechanisms is only 4% slower than when using the original scheduling mechanisms. In addition, the minimum, the mean and the median response times of image data when using our mechanisms are not so different from when not using ours. According to the result, response time of image data pays a small penalty under our scheduling mechanisms.

Therefore, any WWW server that experiences a lot of simultaneous accesses from users would benefit from our resource scheduling mechanisms.

5. RELATED WORK

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

5.1 CPU SCHEDULING

Traditional 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 systems' scheduling policies are usually only available in real-time operating systems, and not in general purpose operating systems in which time-sharing systems' scheduling policies are used. 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 that are higher in priority than regular priorities to real-time applications. Another approach is to schedule based on proportion and/or period [10][11][12]. 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 [13][14][15].

However, none of the above approaches is trying to schedule based on behavior of a process. 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 UNIX based operating systems, several processes of the same priority may be ready to run if they could use the CPU if it were available. Since only one process can be running at a time, the rest will have to wait in the ready queue until the CPU is free and rescheduled on a round-robin basis. In the case of WWW servers, when the number of server processes needed to handle the simultaneous requests increases, if a server process handling an HTML file request is put at the end of the queue, then it takes a longer time for the text data to show up on browsers. As a result, users experience slower response times.

5.2 DISK SCHEDULING

The simplest form of disk scheduling is First Come First Served (FCFS), that schedules requests in the order of their arrival. Since the access schedule thus derived is independent of the relative positions of the requested data on disk, FCFS scheduling can incur significant seek time and rotational latency. Therefore, many scheduling policies concentrated on minimizing seek time such as Shortest Seek Time first (SSTF), SCAN, LOOK, and V(R), and those concentrated on minimizing rotational latency such as Shortest Latency Time First (SLTF) have been proposed in order to achieve higher performance [16][17][18][19]. In other words, these policies attempt to service I/O requests with the minimum mechanical motion. However, they are less concerned about each request individually, which is what our policy does. As a consequence, the problem similar to that when using traditional CPU scheduling policies occurs when using these disk scheduling policies. That is, when a WWW server is accessed by a lot of users simultaneously, the likelihood that an I/O request generated by any server process handling an HTML file request will be put at the end of the queue is high, which results in user experiencing a slower response.

In addition to CPU and disk scheduling, one function in Windows 98, which users can get "faster program start up" as performance enhancement [20], uses an idea similar to ours. 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 load a user's programs faster), but it cannot execute or run a user's programs more efficiently.

6. CONCLUSION

This paper examined the performance of a WWW server when using the proposed CPU and disk scheduling mechanisms. Our scheduling mechanisms controls the allocation of a CPU and a disk drive based on the behavior of WWW server processes rather than based on a fixed policy used in traditional operating systems, in which the utilization of a computer system such as a real-time or a time-sharing system is a major concern. And our experimental result when the WWW server is accessed randomly by multiple requests at the same time shows that by using our disk scheduling mechanism the response time, the time from requesting a WWW page until text data starts displaying, can be reduced. To be more specific, the mean and the median response time are improved 19% and 42% respectively. Moreover, the effect of unfairness due to our policy of giving favorable treatment to server processes handling HTML file requests on the response times of other types of data, which in our case is image data, are relatively small. Therefore, any WWW server that experiences a lot of simultaneous requests from users would benefit from our scheduling mechanisms.

Future work will measure the performance of a WWW server when the operating system's I/O buffer cache in the server machine and each browser's cache are enabled. Also, we will evaluate the usefulness of our idea using other existing software applications.

ACKNOWLEDGEMENT

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

REFERENCES

[1] N. Ford and M. Woodroffe, *Introducing software engineering* (Prentice-Hall, 1994).

[2] R. Pressman, *Software engineering: a practitioner's approach* (McGraw-Hill, 1992).

[3] I. Sommerville, *Software engineering* (Addison-Wesley, 1992).

[4] M. Arlitt and C. Williamson, Internet Web servers: workload characterization and performance implications, *IEEE/ACM Trans. Networking, 5*(5), 1997, 631-645.

[5] C. Cunha, A. Bestavros, and M. Crovella, Characteristics of WWW client-based traces, Technical Report BU-CS-95-010, Computer Science Department, Boston University, 1995.

[6] http://www.apache.org/

[7] S. Suranauwarat and H. Taniguchi, Evaluation of a process scheduling policy for a WWW server based on its contents, *IEICE Trans. Inf. & Syst., E83-D*(9), 2000, 1752-1761.

[8] S. Suranauwarat and H. Taniguchi, A disk scheduling mechanism for a WWW server, *Proc. 2001 International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, 2001. (to appear)

[9] S. Suranauwarat and H. Taniguchi, Operating systems support for the evolution of software: an evaluation using WWW server software, *Proc. 2000 International Symposium on Principles of Software Evolution*, 2000, 292-301.

[10] C. Waldspurger and W. Weihl, Lottery scheduling: flexible scheduling proportional-share resource management, *Proc. 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994, 1-11.

[11] C. Waldspurger and W. Weihl, Stride scheduling: deterministic proportional-share resource management, Technical Report MIT/LCS/TM-528, MIT laboratory for computers science, 1995.

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

[13] 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, 1994.

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

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

[16] H. Deitel, *An introduction to operating systems* (Addison-Wesley, 1990).

[17] A. Silberschatz and P. Galvin, *Operating system concepts* (John Wiley & Sons, 1997).

[18] A. Worthington, Scheduling algorithms for modern disk drives, *Proc. 1994 Conference. on Measurement and Modeling of Computer Systems*, 1994, 241-251.

[19] R. Geist, A continuum of disk scheduling algorithms, *ACM Trans. Comput. Syst., 5*(1), 1987, 77-92.

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

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