九州大学大学院
システム情報科学紀要
第6巻 第1号 平成13年3月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.6, No.1, March 2001

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

Sukanya SURANAUWARAT* and Hideo TANIGUCHI **

(Received December 7, 2000)

Abstract: Conventional process schedulers in operating systems divide a machine's CPU resource among processes using a fixed scheduling policy, in which the utilization of a computer system (e.g., a real-time or a timesharing system) is a major concern rather than content or behavior of a process. As a result, the CPU resource is likely to be used in an inefficient manner, or the processing time of a process might be extended unnecessarily. In this paper, we therefore propose a process' behavior-based scheduler that delays process switching in order to allow the object process to continue its execution even though its timeslice has already expired, when it is predicted from an advanced knowledge called PFS (Program Flow Sequence) that the object process needs a little bit more CPU time before it voluntarily relinquishes the CPU. This allows the processing time or the process switching cost of the object process to be reduced.

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

1. Introduction

Conventional process schedulers divide a machine's CPU resource 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. 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 CPU resource, and it is still used in most workstation operating systems. Real-time systems' scheduling policies [1)~3)] 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 on PCs and workstations has called for new scheduling paradigms to support real-time in systems with conventional timesharing schedulers. One approach to do this is to schedule based on proportion and/or period [4),5)]. 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 [6),7)].

Since the control over the allocation of the CPU resource is based on the utilization of a computer system, none of the above approaches is trying to schedule based on *content* or *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 a timesharing system, each process is assigned a time interval, called its *quantum* or *timeslice*, which it is allowed to run. If the process is still running at the end of its timeslice, 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 timeslice. Because of this, the processing time and the process switching cost increase unnecessarily. For example, when a process uses up its timeslice just before it initiates an I/O operation, it will voluntarily relinquishes the CPU (i.e., the process blocks itself pending the completion of the I/O operation) immediately after the beginning of its next timeslice. 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.

Therefore, we proposed a scheduling idea called POS (Program Oriented Schedule)[8)]. The idea of POS is by increasing an operating system's ability to alter the execution behavior of a program accord-

* Department of Computer Science and Communication Engineering, Graduate Student

** Department of Computer Science and Communication Engineering

ing to the behavior of the corresponding process in the previous execution(s), an operating system could optimize the execution behavior of the program allowing user requirements (e.g., performance enhancement) to be satisfied. By applying this idea to the process scheduler, we can solve the problem mentioned above and also run processes more effectively.

In this paper, we proposed a POS idea-based process scheduler that alters the execution behavior of a program by controlling the timing of process switching, in order to reduce the processing time and the process switching cost. To be more precise, it delays process switching in order to allow the corresponding process to continue its execution even though its timeslice has already expired, when it is predicted from an *advanced knowledge* called *PFS (Program Flow Sequence)* that the corresponding process needs a little bit more CPU time before it voluntarily relinquishes the CPU. The PFS of each program is created based on the behavior of its corresponding process at the end of the first execution, and it is used whenever the program is executed from then on. It also adjusted based on the feedback obtained from each execution. We also present the experimental validation of our scheduler by using a test program with regard to the length of time to delay process switching and the processing time.

2. The Design of Scheduler

A POS 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 used it to create the PFS. If its PFS exists then the process controller will use the PFS to alter the execution behavior of the corresponding process of the program. 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 **Fig. 1**, process A and process B need respectively 3.4 s (seconds) and 2.1 s of CPU time to complete their jobs. Both processes have the same priority and a timeslice of 1 s. **Figure 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 **Fig. 1(a)**, the processing times of process A and process B are 5.5 s and 4.1 s respectively, while in **Fig. 1(b)**, based on the PFS that process B needs only 0.1 s more CPU time to complete its job, the scheduler delays process switching by 0.1 s to allow process B to complete its job. Delaying process switching causes the processing time of process B to be reduced to 3.1 s while that of process A is still the same as in **Fig. 1(a)**. Moreover, the number of process switches decreases from 6 to 4.

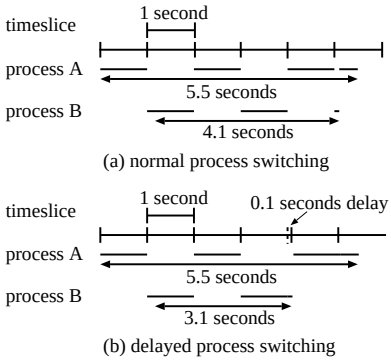

Fig.1 A scheduling example using our scheduler.

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

2.1 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 **Fig. 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 (such as an I/O completion event, for example) before it can proceed.

Next, PFS is created by using the log mentioned

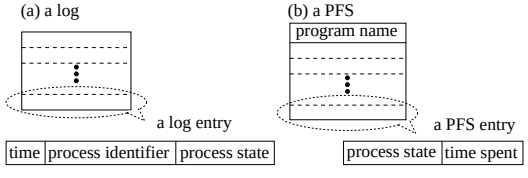

Fig.2 The image of a log and a PFS.

above. A PFS (shown in **Fig. 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 run state as a CPU time of PFS (T_p).

2.2 The Process Controller

When a program is executed, if its PFS exists, then the PFS is used to alter the execution behavior of the corresponding process in order to reduce the processing time and the process switching cost. That is, the PFS is used to control the timing of process switching (e.g., early process switching or delayed process switching) of the corresponding process. In this paper, we discuss only delayed process switching.

Supposed C_c is the current time and C_s is the time that a process starts using the CPU for each allocated portion of CPU; T_e is the expected CPU time a process would use from C_s until it voluntarily relinquishes the CPU (each T_e is determined based on each CPU time of PFS (T_p)) and T_m is the maximum time to delay process switching, called the *maximum dispatch delay time*. When a process is running at the end of its timeslice, the decision about whether the process switching should be delayed or not is determined as follows.

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

(2) If $T_e - (C_c - C_s) > T_m$, then the next waiting process is dispatched.

According to this, when a process is running at the end of its timeslice, 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 is not always the same every time the program is executed (i.e., it varies little by little whenever the program is executed), PFS needs to be able to adjust itself to changes. However, adjusting PFS to the latest change is dangerous when the corresponding process runs abnormally. So we adjust 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.

(1) If $T_p = (C_c - C_s)$, then the adjustment is not needed.

(2) If $T_p < (C_c - C_s)$, then T_p should be increased by using the following formula:
$T_p = T_p + \{(C_c - C_s) - T_p\} \times (x/100)$, (1)
where x is an increase scaling factor (%).

(3) If $T_p > (C_c - C_s)$, then T_p should be reduced by using the following formula:
$T_p = T_p - \{T_p - (C_c - C_s)\} \times (y/100)$, (2)
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 T_p is increased or decreased slightly by using an increase or a decrease scaling factor.

3. Implementation

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

3.1 The Logger

The main work of the logger is to create a log and a PFS. A log (shown in **Fig. 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 **Fig. 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.

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.

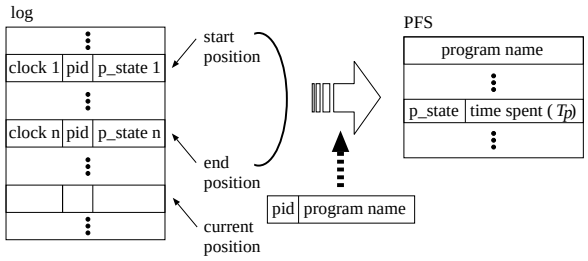

Fig.3 The diagram of how to use the log to create a PFS.

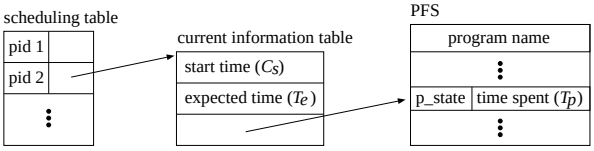

Fig.4 The diagram of how to schedule a process based on PFS.

- 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 formula where C_i is a *clock i* element in the log entries in that series.

$$\begin{aligned} T_p = \quad & \{C_2(ready) - C_1(run)\} \\ & \vdots \\ & + \{C_n(ready) - C_{n-1}(run)\} \\ & + \{C_{n+2}(wait) - C_{n+1}(run)\} \end{aligned}$$

3.2 The Process Controller

The main work of the process controller is to schedule 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 timeslice, (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 **Fig. 4** and **Fig. 5** respectively.

I. When a process is created,
   if the PFS of the program it run exists, then
   (1) its process identifier is stored in a table called the *scheduling table* and space for a table called *current information table* is allocated. A scheduling table (shown in **Fig. 4**) is an array of structures containing process identifier (*pid*) of the process which will be scheduled based on PFS and a pointer to its current information table. A current information table (shown in **Fig. 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).
   (2) 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.

II. When a process uses up its timeslice,
   (1) the elements of the currrent 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.

(2) Next, the process is scheduled according to the following rule.

- 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 $T_e < 0$.

III. When a process blocks itself,

(1) the element of PFS entry, T_p, is adjusted according to the following rule.

- 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 $T_p \leftarrow T_p - \{T_e - (C_c - C_s)\} \times (y/100)$.

(2) 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.

IV. When a process terminates,

(1) the element of PFS entry, T_p, is adjusted according to the following rule.

- 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 $T_p \leftarrow T_p - \{T_e - (C_c - C_s)\} \times (y/100)$.

(2) 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 $C_s \leftarrow C_c$.

4. Experimental Evaluation

Our scheduler is implemented in BSD/OS 2.1. The experiments were run on a 120 MHz Pentium with 32 MB of memory, running our modified version of BSD/OS 2.1. All our experiments were conducted in single user mode with the preemption enabled. Also, timeslot and timeslice are 1 and 100 ms (milliseconds) respectively.

We used a test program to evaluate the effectiveness of our scheduler with regard to the overhead, the relation between the length of the maximum

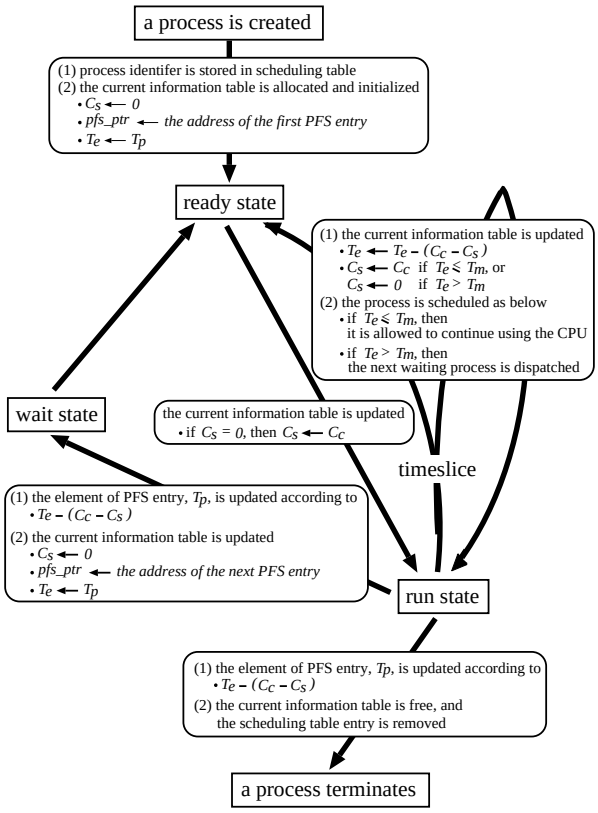

Fig.5 The processing flowchart.

dispatch delay time and the processing time.

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

4.1 Overhead

This section presents the experimental results for the overhead involved in our scheduler, i.e., the cost involved in creating and storing PFS on memory and the cost involved in scheduling based on PFS, when running the test program.

(1) 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.

(2) 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.

4.2 The length of the Maximum Dispatch Delay Time vs. The Processing Time

We ran the test program with various arguments specifying the amount of time for work A as 75, 125, 150 and 200 ms and found the relation between the length of the maximum dispatch delay time and the processing time. In order to enable process switching when a given timeslice expires, throughout the experiment, the test program coexisted with the *loop program*, i.e., the program that loops only work A. **Figure 6** shows the experimental results plotted with the processing time on y-axis normalized by the processing time when using a conventional timesharing scheduler. This figure shows that when the required CPU time is more than the timeslice (100 ms) and its difference is smaller than the maximum dispatch delay time, then the processing time becomes shorter. For example, when the required CPU time is 125 ms and the maximum dispatch delay time is 40 ms; as well as when the required CPU time is 125 and 150 ms and the maximum dispatch delay time is 70 ms, the processing time becomes shorter. This verifies that our scheduler works as expected, i.e., it delays process switching as expected.

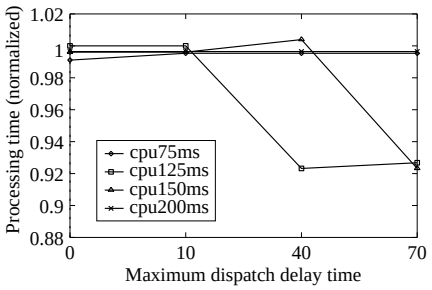

Fig.6 The relation between the length of the maximum dispatch delay time and the processing time (in case of the test program).

5. 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 dispatch and then creates an advanced knowledge called PFS (Program Flow Sequence), and (2) when the program is executed from then on, it schedules the corresponding process by using 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.

The cost involved in our scheduler is small. And the experimental results with the test program show that our scheduler delays process switching as expected and the processing time can be reduced by using it.

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

References

1) W. Shih, J. Liu, and C. Liu, "Modified Rate-monotonic Algorithm for Scheduling Periodic Jobs with Deferred Deadlines," IEEE Trans. Software Eng., 19(12):1171–1179, 1993.
2) A. Burns, K. Tindell, and A. Wellings, "Effective Analysis for Engineering Real-time Fixed Priority Schedulers," IEEE Trans. Software Eng., 21(5):475–479, 1995.
3) W. Feng and J. Liu, "Algorithms for Scheduling Real-time Tasks with Input Error and End-to-end Deadlines," IEEE Trans. Software Eng., 23(2):93–106, 1997.
4) C. Waldspurger and W. Weihl, "Stride Scheduling: Deterministic Proportional-share Resource Management," Tech. Rep. MIT/LCS/TM-528, MIT laboratory for computers science, 1995.
5) M. Jones, D. Rosu, and M. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," Proc. of the 16th ACM Symposium on Operating Systems Principles, pp.198–211, 1997.
6) B. Ford and S. Susarla, "CPU Inheritance Scheduling," Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation, pp.91–106, 1996.
7) P. Goyal, X. Guo, and H. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation, pp.107–121, 1996.
8) H. Taniguchi, "POS: Program Oriented Schedule," Proc. of the 1996 IPSJ Computer System Symposium, pp.123–130, 1996. (in Japanese)