Vol. 43 No. SIG 6(HPS 5) IPSJ Transactions on High Performance Computing Systems Sep. 2002

Regular Paper

Priority Enhanced Stride Scheduling

Damien Le Moal,[†,☆] Masahiro Ikumo,[†] Tomoaki Tsumura,[†]
Masahiro Goshima,[†] Shin-ichiro Mori,[†] Yasuhiko Nakashima,[†]
Toshiaki Kitamura[†] and Shinji Tomita[†]

Whereas classical scheduling methods like priority scheduling can efficiently support the execution of various type of applications, fair-share scheduling methods do not provide good results for I/O bound and interactive processes execution. This paper presents a novel implementation of the fair-share scheduling method called stride scheduling and its extension using a more classical priority scheduler to support both compute bound and interactive applications. Evaluation results show that it improves fair-share allocation of CPU time among users and processes compared to strict fair-share scheduling methods and that the execution of interactive processes is not degraded. It is also shown that the scheduling overhead is bounded and does not depend on the number of runnable processes.

1. Introduction

We have been doing a research on the Computer Colony, a cluster of physically distributed computers behaves like a single system [7)] if necessary for parallel processing, while preserving the autonomy of each computers. Not like a rack mounted PC cluster for application services, individual computer nodes of the Computer Colony may also work as a ordinal desktop computer for programming, text editing, Web browsing and so forth, executing some parallel and/or time consuming applications in back ground. Thus, a new scheduling method is needed to support such workload and especially to implement a fair-share allocation of processing time among users, without degrading the respondability to interactive and I/O intensive processes.

Recent research works have produced several specialized scheduling method implementing a fair-share allocation of processing time like stride-scheduling. However, most of these only perform well for compute bound applications, degrading the response time of interactive and I/O bound processes.

In this paper, we propose a fair-share CPU scheduling method based on a new implementation of stride scheduling reducing the overhead of fair-share allocation of cpu time. This implementation is enhanced with a classical priority scheduler which can provide the desired control over process scheduling in order to efficiently executes interactive processes and also to improve the overall fairness.

The rest of this paper is organized as follows. Section 2 discusses background related to our scheduling scheme. Section 3 presents the proposed scheduling algorhithm and its implementation. Section 4 shows some experiment results, and section 5 concludes this paper.

2. Background

Operating systems tries to execute various type of applications efficiently by using general scheduling policies. Such policies may however favor one class of processes over another. Priority scheduling is certainly the most commonly used policy because it can implement several policies with the same mechanism. Some operating systems also uses more simple schemes like the round robin policy. The following sections briefly discusses some general scheduling policies which consider fair-share allocation, and then mentions about some related works in more detail.

2.1 Round-Robin Scheduling

A simple approach of the process scheduling problem is the Round-Robin policy (RR). This policy defines a small time unit called *time quantum* (or *time slice*). Processes are chosen in the order of arrival and allowed to run only for one time quantum. If a process do not complete its CPU burst during this time, it is put back at the end of the ready queue and another process if one is available is chosen.

The perfomance of this policy depends greatly on the length of the time quantum. If

† Kyoto University
☆ Presently with Systems Development Lab., Hitachi Ltd.

it is too long, RR behaves like First Come First Service (FCFS) and shows poor performance results for I/O bound processes, whereas turn around time of compute bound processes can be improved. On the contrary, if the time slice is too short, response time is improved by getting closer to the results provided by the Short Job First (SJF) policy. From the user point of view, if the time quantum is too short, this policy is seen as if each of the n runnable processes are each executed on their own processor running at $1/n$ the speed of the real processor, thus it is called *processor sharing*. However, processor sharing is never used because the ratio of time spent performing context switches becomes very high, increasing system overhead.

2.2 Priority Scheduling

Priority scheduling associates a priority (that defines an order of importance) to each process. The process with the highest priority is always chosen first for execution. Other policies are needed to make a choice between processes of the same priority. The resulting scheme is the more general multilevel feedback queue scheduling policy, where the priority scheduling being used between queues and processes of equal priority are scheduled using different policies.

Several methods can be used to calculate priorities, depending on various parameters. A widely used scheme calculates priorities depending on the run-time behavior of processes. In particular an approximation of the SJF policy that can be implemented by assigning a high priority and a small time quantum to processes which often release the processor voluntary to perform I/O, whereas compute bound processes having a lot of long CPU bursts are assigned a low priority and a longer time quantum.

Classes of priorities can also be implemented depending on the type of processes. For example, the SVR4 version of the Unix operating system defines a real-time priority class 9),19) for processes with time constraints, a system priority class for system processes and the lowest priority class for users applications.

However, a major problem of the priority scheduling policy is that none of the runnable processes is guaranteed to receive some processor time in a finite time interval. This problem known as the *CPU starvation* problem can be solved simply by recomputing regularly process priorities, which can greatly increase the overhead of the system.

2.3 Fair-share Scheduling

Systems guaranteeing that the CPU starvation problem will not occur are usually said to be fair. However, a user is not guaranteed at all to receive as much CPU time as another one. In fact, all the scheduling policies presented previously will allocate more CPU time to users running several processes at the expense of users running only few processes.

Fair-share scheduling (or proportional fair-share) policies try to address this problem by scheduling processes using the share of CPU time defined for a user or his processes. Shares are generally calculated using a currency, for example *tickets* so that a user or a process with t tickets in a system with a total of T tickets is allocated t/T of the processing time. A hierarchical definition of tickets and share is also possible to define both a user share and a process share.

2.3.1 Priority-based Fair-share Scheduling

The first fair-share schedulers were priority based. In this scheme, priorities are calculated depending on both the share of CPU time allocated to users or processes and the past CPU usage. Famous works in this area are the fair-share scheduler of AT&T 15), the work presented in Ref. 8) or the event based fair-share scheduler described in Ref. 6).

Priority-based fair-share schedulers can provide good results, but usually realize a fair allocation of CPU time over long time intervals. Moreover, their priority calculation methods are generally difficult so that it makes such scheme inflexible and difficult to tune. The work presented in Ref. 8) recomputes all process priorities every 4 seconds, with fairness realized over hours or days. In the work presented in Ref. 6), priorities are recalculated on event occurrence. In this case, since one event (I/O completion, preemption, ...) only concerns one or few processes in general, the overhead of priority calculation can be reduced. This approach is called event-based scheduling, and provides better results for fairness. However, it is not precise because it must consider a lot of parameters to penalize or boost processes execution depending on their CPU usage.

2.3.2 Lottery Scheduling

More recent works in the area of fair-share scheduling have produced several interesting methods, like lottery scheduling 16). In lottery scheduling, each process is assigned a number of tickets proportional to its share. The scheduler selects a ticket randomly using a lottery and the process owning the selected ticket is executed for one time quantum.

However, lottery scheduling only provides a probabilistic fairness over a certain time interval. Also, as the run time behavior of a process is not considered at all, in its basic implementation, lottery scheduling performs poorly for the execution of interactive and I/O bound processes. The dispatch latency on wake up☆ under this policy depends highly on the number of tickets owned by a process and may be longer compared with the priority-based approarches.

2.3.3 Stride Scheduling

Stride scheduling[17)] is a deterministic fair-share scheduling scheme using an algorithm similar to rate-based network flow control algorithms. Like lottery scheduling, tickets are used to define the share of CPU time allocated to a process.

In stride scheduling, the basic idea is to compute the time (or stride) a process must wait before receiving its next time quantum. The stride of a process is thus inversely proportional to its share: the higher the share is, the shorter the stride is. A pass parameter is associated to each process and incremented by its stride each time it is scheduled. The scheduler always chooses the process with the smallest pass for execution.

Figure 1 shows an example where three processes (process A to C) are allocated respectively 200, 300 and 500 tickets, resulting in shares of 20%, 30% and 50%. Process strides calculated as $3/share$ are respectively 15, 10 and 6. At start time, all processes pass are null. Processes are thus first scheduled in the A, B, C order. At this moment, since process C has the smallest pass (6), it is scheduled agin. Now, the pass value of process B becomes smallest, then process B is scheduled next. After that process C is scheduled agin. Over the sequence of this example in Fig. 1, processes A, B, and C are scheduled twice, three times and five times, respectively, resulting in the desired $2:3:5$ ratio. At the end of this sequence, all processes have the same pass value, the same scheduling pattern will then be repeated again. Stride scheduling thus allow to achieve fairness over very short time intervals.

☆ The dispatch latency on wake up stands for the latency between the time a process becomes runnable maybe after the completion of the requested I/O and the time this process resumes its execution.

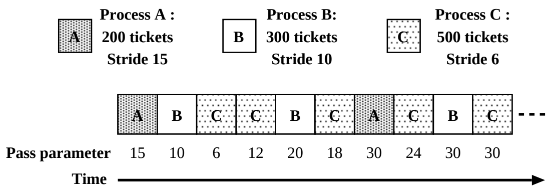
Process A : 200 tickets Stride 15
Process B: 300 tickets Stride 10
Process C : 500 tickets Stride 6
A B C C B C A C B C
Pass parameter 15 10 6 12 20 18 30 24 30 30
Time

Fig. 1 Stride scheduling.

To allow a great precision of this mechanism even for very small differences of process shares, the authors proposed a simple method for stride calculation. Even if the stride of each process can be defined as a floating point value, for simplicity an integer type was used. The stride of each process is calculated as follows.

$$process\ stride = rint\left(\frac{stride\ constant}{process\ share}\right)$$

Where `rint()` is the function rounding a floating point to the nearest integer. In the original paper, the stride constant was 2^{20} which is big enough so that the process stride value round error to the nearest integer becomes negligible compared to the stride value itself, making such error practically undetectable.

The ready queue structure used here is a sorted list in increasing pass order. The resulting overhead of this method is thus not negligible for systems with a high load. At best this overhead is in $O(log(n))$, where n is the number of processes, depending on the sort algorithm. Moreover, as stride scheduling does not consider the run-time behavior of processes, it suffers from several problems, as same as lottery scheduling.

- The dispatch latency on wake up depends highly on the share of CPU a process was alloted. Indeed, to preserve fairness, both stride and lottery scheduling are not preemptive, so that a process waking up after waiting for I/O completion cannot preempt the currently running process.
- The straightforward implementation of stride scheduling (and lottery scheduling) implements only a strict fair-share allocation of CPU time. As a result, a process sleeping waiting for I/O completion cannot be allocated its fair-share of CPU time because the time slept is lost and is not compensated by allocating more CPU time on wake-up.

Several modifications of the basic algorithm of both stride and lottery scheduling are possible to improve fairness and responsiveness. These improvements are presented in the Section 3.

2.4 Related Works

Recent works (presented in Refs. 2) and 14)) have added support to improve the interactivness of strict fair-share scheduling methods. These modifications are mainly based on dynamic share adjustment using ticket allocation.

As for stride scheduling, an improvement presented in Ref. 2) uses exhausting tickets (tickets whose validity is timely limited) with a loan and borrow mechanism. With this extension, runnable processes can borrow the tickets of the sleeping ones to increase their share. But they must pay back the lent processes when they wake up, increasing their share and thus their chances to be scheduled quickly. Another more simple method is based on system credits: where the system gives more tickets to a sleeping process in proportion to the slept time. Another improvement presented in Ref. 17) uses non-uniform time quantum. A process using a fraction $f < 1$ of its time quantum has its pass increased by only $f * pass$. As interactive and I/O bound processes generally do not use all their time quantum, their pass are less increased so that they are scheduled sooner. However, all these methods do not implement a preemptive scheme and they increase the dispatch latency of processes on wake up. Also, if there are a lot of interactive processes, they cannot anyway be handled efficiently because interactive processes' CPU burst is usually too short compared to the time needed to benefit from those dynamic ticket adjustments. The overhead introduced is also not negligible.

Some enhancement of the basic lottery scheduling method uses similar methods. If a process sleeps waiting for I/O, it is granted a timely limited ticket boost on wake up, increasing its chances to be scheduled quickly. The work presented in Ref. 14) uses this modification. It combines the system level priorities of the FreeBSD scheduler 11) with lottery scheduling. The response time of interactive processes can thus be minimized by scheduling them depending on their priority on wake up, whereas user level processes are assigned the lowest priority and chosen with lottery scheduling. This method thus improve both fairness and responsiveness. However, the difficult dynamic adjustment of tickets allocation increases the total overhead and makes its implementation difficult to tune.

Recently, many works 1),3),5),12),13),18) have been done on fair-share or QoS scheduling for server like environment, but most of these do not concern interactive jobs. Among those, remarkable works for us are the Borrowed-Virtual-Time (BVT) scheduling 5) and the Virtual-Time Round-Robin (VTRR) scheduling 13). Both of the BVT and the VTRR schedulings are based on the concept of virtual time, a complemental concept to the pass in stride scheduling, where the virtual time of each process advances inversely proportional to its share when they spend real CPU time and the process with minimum virtual time is selected for execution.

The BVT scheduling was proposed almost same time as our first proposal 10). This scheme provides low latency dispatch of latency sensitive processes, while maintaining middle term fair-share, thus the behaviour of the BVT scheduler can be quite similar to ours. For this purpose, latency sensitive processes may borrow some amount of virtual time and use it to rewind their own virtual time so that they gain dispatch preference. This mechanism has a similar effect as our boosting scheme mentioned in Section 3.2 but it differs in a sense that the borrowed virtual time will never be returned and that the only explicitly specified processes may have a chance of this time warping.

The VTRR scheduler is interesting in that it realizes $O(1)$ scheduling overhead like ours. The VTRR scheduler combines the benefit of low overhead round-robin scheduling with the high accuracy mechanisms of virtual time (VT) and virtual finishing time (VFT) used in fair queueing algorithm 4). The data structure used in the VTRR scheduler is a simple doubly-linked list in which runnable processes are stored in descending order of share. A selection of the next process is done in round-robin manner as long as the VFT of the candidate process is earlier than that of the previous process. If an VFT inversion occurs, the scheduler skips the remaining processes in the linked list and starts selecting processes from the beginning of the list. Such a simple selection mechanism make VTRR possible to realize $O(1)$ selection time but its fairness is achieved only after relatively long period. And as well as other recent work on fair share scheduling, VTRR scheme is designed for server like environment so that it neither considers latency sensitivity nor supports preemption.

3. Enhanced Stride Scheduling

Extending fair-share scheduling using priority scheduling can provide the desired control over processes execution order while fairly allocating processor time among users and processes. However, in its basic implementation, stride scheduling does not support process preemption and has a high overhead. This section first presents our implementation of stride scheduling which solves these problems. The overall CPU time allocation policy to improve both fairness and responsiveness is then discussed. Finally, the scheduling algorithm implementing this policy is presented in more detail.

3.1 Stride Scheduling Implementation

This section presents our implementation of stride scheduling. It avoids sorting processes in pass order and is also modified to support process preemption without degrading the strict fair-share allocation of CPU time provided by the basic algorithm.

The data structure used to choose processes is a circular array of linked lists. Each list contains processes with equal pass. A pointer called “head” indicates the list of processes with the smallest pass. All dequeue operations are thus done in the head list. **Figure 2** shows how the example of Fig. 1 is scheduled using this data structure. Three states of the stride queue are represented: the initial state, the state after three schedule operations and after nine schedule operations.

The head pointer is forwarded to the next non-empty list each time the head list becomes empty. A process consuming all its time quantum without sleeping is put back at the end of the list with an index equal to the current head list increased by the current stride of this process. Thus, the circular array indexes are used as a pass parameter, moving a process into a higher index list virtually increments this process pass by its stride. The pass parameter is not needed any more and sorting processes is avoided as lists are naturally ordered in increasing pass order from the list pointed to by the head.

A different enqueue operation is used for preempted processes. Such process are inserted at the beginning of the current head list, and their time quantum is set to the remaining time of the previous run. Doing so, the relative share

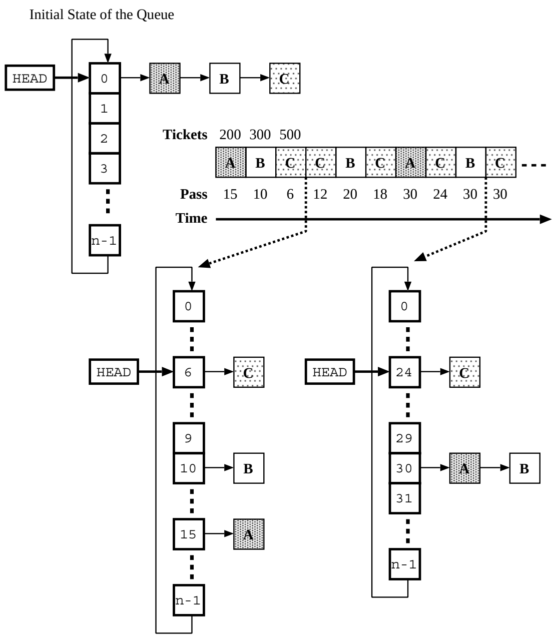

Fig. 2 Example of Fig. 1 using our implementation of stride scheduling.

allocation of CPU time between processes in the stride queue can be preserved.

The stride calculation proposed by the authors of stride scheduling generates big integers. When the stride is calculated using the defined share, the rounding error becomes negligible, allowing a great precision. Such stride values cannot be practically used to calculate an array index. Smaller values of processes stride, but yet precise, are necessary.

We simply calculate the stride of a process using the formula

$process\ stride = rint(1/process\ share)$

where $rint()$ is the function rounding a floating point to the nearest integer. As shown in **Fig. 3**, the error generated by the rounding operation is recorded and increased by the initial error each time a process is put back into the stride queue. If the total error becomes higher than one, the displacement into the stride queue defined by the stride is increased by one.

As the stride queue size is fixed, a resolution problem can appear for processes with a stride bigger than the stride queue size (i.e., with a small share): all these processes will get a stride equal to the size of the stride queue, thus the same share. This size must then be big enough to support very small shares precisely. The size of the stride queue is 1,024 so that the possible minimal share is equal to $1/1{,}024 \approx 0.001 =$

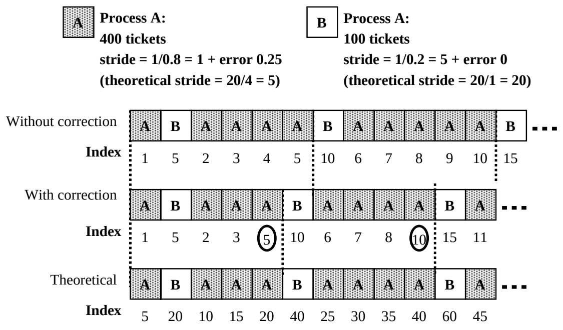
Process A:
400 tickets
stride = 1/0.8 = 1 + error 0.25
(theoretical stride = 20/4 = 5)
Process A:
100 tickets
stride = 1/0.2 = 5 + error 0
(theoretical stride = 20/1 = 20)
Without correction
Index
With correction
Index
Theoretical
Index

Fig. 3 Effect of the stride calculation method on process scheduling. The circled index values are incremented with the stride plus the correction.

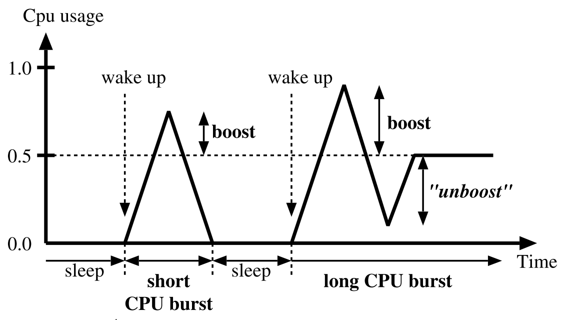

Fig. 4 I/O bound processes must be boosted on wake up. When having a longer CPU burst, the boost must be compensated to preserve the overall fairness.

0.1%.

This implementation reduces stride scheduling overhead but still only provides a strict fairness and do not consider the run-time behavior of processes. A more flexible CPU time allocation policy is needed. This matter is discussed in the next section.

3.2 Scheduling Policy

Processes waking up after waiting for I/O completion should be scheduled quickly and allowed to receive temporarily more than their defined share. Moreover, the scheduling policy should support preemption to minimize the wait time in ready queue on wake up. Such behavior is desirable to emulate the short-job-first policy by allowing a process to complete a short CPU burst ahead of other processes. It can also improve kernel contention as kernel shared resources can be released quickly.

In such cases, the allocated CPU time may be higher than the defined share of a process on a certain interval of time. Depending on the past CPU usage, basically the following two cases should be considered.

- The process completes its CPU burst within the allocated share on wake up and sleeps before consuming more than its defined share.
- The process has a longer CPU burst and is still runnable after receiving its defined share on wake up. In this case, the boost as a result of the over share allocation should be compensated with a "*unboost*" to preserve the overall fairness. However, other processes should not be penalized for having used the share of processor time unused by sleeping processes.

To be reactive to such situations, the scheduler should not only consider the total CPU

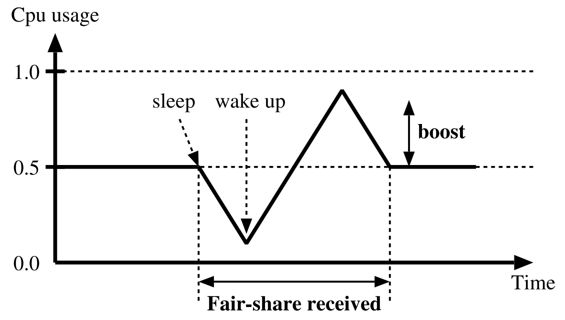

Fig. 5 A process having a long CPU burst and sleeping only very short intervals can still be allocated its defined share.

time allocation over the life time of a process, but also should measure the recently allocated share over shorter intervals. An accounting window recording the past CPU allocation of a process can be used for this purpose. The allocated share $a_w(t)$ of a process at time t over a window of time length w can be calculated simply with the following formula.

$$a_w(t) = \frac{CPU\ time\ allocated\ from\ (t-w)\ to\ t}{w}$$

Figure 4 shows the allocated share of an hypothetical I/O bound process measured over such accounting window. The two cases discussed above are represented.

Measuring the allocated share over the accounting window can provide a precise mechanism to control process boosts and compensations. As a result, slept time shorter than the window length can be easily overlapped by a not compensated boost, allowing processes sleeping for I/O to receive their fair share. **Figure 5** shows such behavior. The allocated share of the represented process decreases as the process sleeps. On wake up, it is boosted without compensation. The resulting overall share allocation matches the defined share.

However, such mechanism should be precisely controlled to avoid a process sleeping a long time to be boosted inconsiderably on wake up. In fact as stated above, in such case, the boost received on wake up should be compensated such that other processes are not penalized for having used the share of processor time unused by sleeping processes.

The next section presents in more detail the scheduling algorithm which implements this policy.

3.3 Priority and Stride Scheduling Combination

The core idea is to combine a classical priority scheduler with our implementation of stride scheduling. Priority scheduling allows to simply implement a preemption control mechanism on wake up. It also allows to implement the boost on wake up by scheduling a process with a high priority until it is allocated its defined share. Stride scheduling can implement the fair allocation of CPU time without any difficult priority calculation method. Its deterministic behavior also allows a precise control of the process "*un-boost.*"

The ready queue has a classical multilevel feedback queue structure, as shown in **Fig. 6**. The stride scheduling queue presented in Fig. 2 is assigned the second lowest priority (priority 1). All other priority levels use a RR policy with different time quantum. Processes are chosen in decreasing priority order. On wake up, a process is always assigned a priority higher than 1 so that it is scheduled ahead of all processes in the stride queue. Stride scheduling is used only if no process has a higher priority than the stride queue.

Figure 7 shows the table used as a base to calculate priorities and time quantums[☆]. For each priority level, this table defines 1) the next priority `Pexp` when a process uses all its time quantum, 2) the priority on wake up `Pslp` if a process sleeps waiting for an event from this priority level and 3) finally the time quantum `TQ` for this level. If a process is preempted before using up its time slice, its priority is decreased depending on the used ratio of its time quantum so that processes tend to stay more priority levels.

☆ This table was based on Solaris 2.5 but with some modifications through trial and error in our experimental environment.

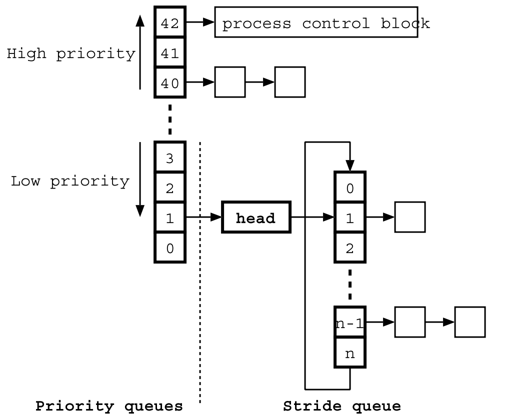

Fig. 6 The ready queue is a classical multilevel feedback queue structure using priority scheduling between queues. The stride queue is assigned a fixed priority (priority 1).

```
int pritbl[] = {

        // PRIO      TQ       Pexp      Pslp

        /*  0 */     100,      0,        0,       // Kernel threads

        /*  1 */     100,      1,        32,      // Stride queue

        /*  2 */     100,      2,        33,      // normal priority queue
        /*  3 */     100,      2,        34,
        /*  4 */     100,      2,        35,
       ---
        /*  9 */     100,      2,        40,
        /* 10 */     100,      2,        41,
        /* 11 */     100,      2,        42,

        /* 12 */      80,      2,        42,
        /* 13 */      80,      3,        42,
       ---
        /* 19 */      80,      9,        42,
        /* 20 */      80,     10,        42,
        /* 21 */      80,     11,        42,
        /* 22 */      60,     12,        42,
        /* 23 */      60,     13,        42,
        /* 24 */      60,     14,        42,
       ---
        /* 30 */      60,     20,        42,
        /* 31 */      60,     21,        42,
        /* 32 */      40,     22,        42,
        /* 33 */      40,     23,        42,
        /* 34 */      40,     24,        42,
       ---
        /* 39 */      40,     29,        42,
        /* 40 */      40,     30,        42,
        /* 41 */      40,     31,        42,
        /* 42 */      20,     32,        42
```

Fig. 7 Priority table.

High priority processes are assigned a short time quantum (for example, 20 ms) when scheduled. This time quantum is longer for lower priority levels and is set to 100 ms for the stride scheduling queue level. By assigning a high priority to processes on wake up, this method allows a good emulation of the short-job-first policy which is probably optimal for interactive systems.

The scheduler recomputes priorities on event occurrence, when processes are preempted either at the end of their time quantum by the last clock tick or by a higher priority process waking up. Unlike priority based fair-share scheduler, the calculation method used is simple. The accounting window used to precisely measure CPU time allocation is separated in

Table 1 Summary of the scheduler action.

Current Priority(P)	Allocated Share (S_A)	Used Time Quantum (TQ_{used})	Event	Next Priority	Next Time Quantum	Rem.
$P \neq 1$ (Priority Queue)	$S_A \geq S_D$	—	—	1	$TQ_1 \times$ penalty	†1
	$S_A < S_D$	$TQ_{used} = TQ_P$	—	$P_{exp}(P)$	$TQ_{P_{exp}(P)}$	
		$TQ_{used} < TQ_P$	Sleep	$P_{slp}(P)$	$TQ_{P_{slp}(P)}$	
			Preemption	$P_{pre}(P,TQ_{used})$	$TQ_{P_{pre}(P,TQ_{used})}$	
$P = 1$ (Stride Queue)	$S_A \geq S_D$	TQ_1'	—	1	$TQ_1\times$ penalty	†2
	$S_A < S_D$	TQ_1'	—	1	TQ_1	
	—	$TQ_{used} < TQ_1$'	Sleep	$P_{slp}(1)$	$TQ_{P_{slp}(1)}$	†3
			Preemption	1	TQ_1 - TQ_{used}	†4

†1 : Move to Stride Queue with unboosting, †2 : unboost, †3 : Move to Priority Queue,
†4 : Position in the stride queue unchanged.

S_D : defined share

P_{pre}(P,TQ$_{used}$) $= P - \mathrm{int}[(P - P_{exp}(P)) \times \frac{TQ_{used}}{TQ_P}]$

penalty : Assuming that the sweeping window is logically divided into N sections, currently we choose N as 4, boosting effect in one section is going to be compensated during the succeeding $N-1$ sections by means of reducing the time quantum during these sections with the following penalty function.

$$penalty = f(S_D, S_A, N) = max\{0, \frac{N\times S_D - S_A}{(N-1)\times S_D}\}, \qquad (0 \leq penalty \leq 1.0)$$

$$\mathbf{TQ_1'} = \begin{cases} TQ_1 & \text{(ordinary section)} \\ TQ_1\times \text{penalty} & \text{(unboosting section)} \end{cases}$$

two intervals of equal length. Depending on the CPU allocation measured over the most recent interval, the following two cases can occur.

- If the preempted process was allocated less than its defined share over the most recent interval, its priority is decreased using the priority table depending only on its current priority, and on its time quantum usage. In this case, the priority is never decreased to 1.
- On the contrary, if the preempted process was allocated its fair-share or more, its priority is set to 1 so that it will be scheduled using stride scheduling from the next time.

A boosting process will thus be scheduled ahead of processes in the stride queue using priority scheduling until it is allocated its fair-share. Then, a process can receive the processing time consecutively to match his defined share. This can result in a over share allocation over the entire window when the process enters the stride queue. If a process is still runnable when entering the stride queue, its time quantum may be decreased depending on the past CPU allocation to implement the "*unboost*". This penalty will be thus recorded in the most recent interval of the window, the boost recorded being shifted to the end of the window. Doing so, fairness can be precisely controlled over the total window.

However, with this scheme, I/O bound processes which always release the CPU before the end of their time quantum may be able to use more than their defined share. As a solution to this problem, a control point is added on return from system call, where a process is about to return to user space. If the allocated share is higher than the defined share, the process is preempted as soon as it returns to user space, and its priority is decreased to 1. This also prevents a process from boosting several times in a short interval, which may happen if the process always sleeps just at the end of its boost while it is scheduled in the priority queue.

The overview of our scheduler action is summarized a little bit in detail in **Table 1**. The next section presents some evaluation results explaining in more detail the effect of the window time length on the scheduler behavior.

4. Evaluation Results

The results presented in this section were measured using an emulator of a distributed operating system actually developed in our laboratory. This emulator simulates program execution state change depending on parameters defining the distribution of CPU and I/O bursts of processes. First we discuss the scheduling overhead and then we argue the feature of our scheduler showing some experimental results.

4.1 Scheduling Overhead

We have compared the relative overhead of stride scheduling queue manipulation for our implementation and for the original implementation proposed in Ref. 17). **Figure 8** shows the cost of put and get operations for both implementations depending on the number of runnable processes. Those measurements were

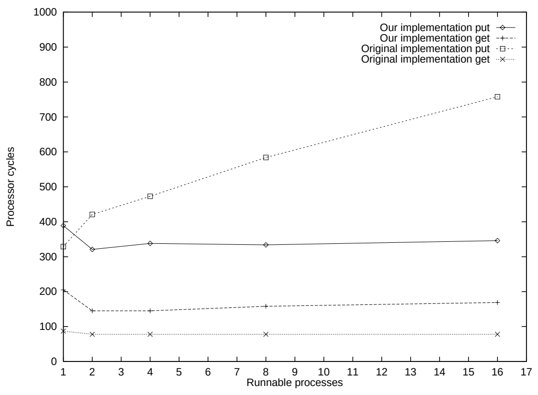

Fig. 8 Stride queue manipulation cost (put and get operations).

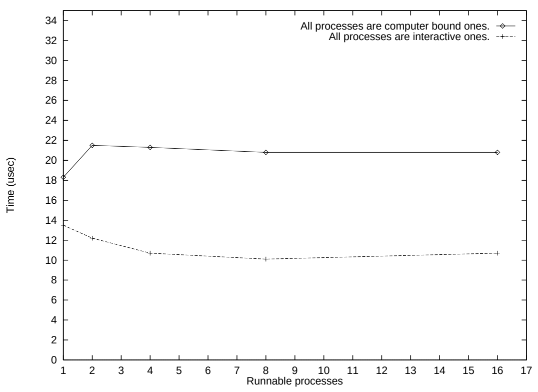

Fig. 9 Scheduling overhead.

done on a 300 MHz Ultra-Sparc II machine. Our implementation provides a bounded "put" operation cost, whereas in the original implementation the "put" cost increases with n because of queue search. However, the get operation is slightly slower because of the head pointer update in the stride queue structure.

Figure 9 shows the cost for the execution of the `schedule()` function which chooses the next process. This function updates CPU usage information of the preempted process, recalculates the priority of this process and puts this process back into the ready queue if it is still runnable. If the process is put back into the stride queue its stride is recomputed. `schedule()` then chooses the next process to execute and update its accounting window with the waited time.

The graph shows the scheduling overhead of two cases. The one is when only priority scheduling is used and the other is when there are only compute bound processes and stride scheduling is used. The overhead of both priority and stride scheduling do not depend on the number of runnable processes. In the average, the scheduling overhead will thus be bounded with these two limits when both methods are used. On heavily loaded systems, it will tend to be closer to stride scheduling overhead as the stride queue may be more intensively used.

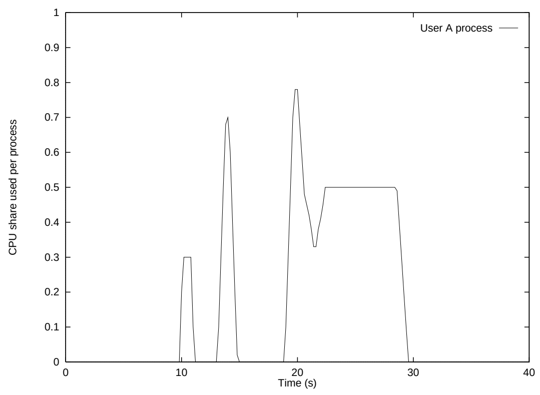

Fig. 10 Basic scheduling behavior for different CPU bursts length.

4.2 Experiments

In this section, we show that the proposed scheduling mechanism allows a precise allocation of CPU time among users and processes. First some characteristic cases are considered to show the basic behavior of the scheduler. This is followed by more general tests for compute bound and I/O bound processes. Note that, in these experiments, simple working sets are chosen deliberately so that the resulting scheduler actions are analyzable.

4.2.1 Basic Behavior

This section demonstrates how the boost/unboost mechanism works in our scheduler. **Figure 10** shows the basic behavior of the scheduler. In this example, a user A with a defined share of 50% runs a process which has three CPU bursts of length 300 milliseconds, 700 milliseconds and 5 seconds, each. Another user B is running two compute bound jobs (not represented in the figure). The accounting window length is 2 seconds, and the graph shows results measured over one second intervals.

As the two first CPU bursts of user A process are short, they are executed within the allocated CPU time with priority scheduling, ahead of user B processes. Because it sleeps quickly and a long time, user A process is allowed to be boosted also for the next CPU burst. However, as this last CPU burst is longer, the boost increases. As a result, user A process is penal-

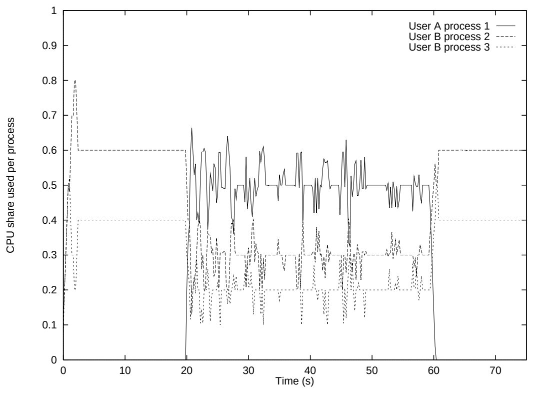

Fig. 11 Two users A and B with the same defined share run compute bound processes. User A process starts after 20 seconds.

ized to preserve fairness. When the CPU time allocation over the window of two seconds is restored, the process is scheduled with the stride scheduling until it completes.

4.2.2 Compute Bound Processes

This section illustrates how fairness is improved by the compensation of the time slept by processes with the boost on wake up.

In the example of **Fig. 11**, user A is running one compute bound process starting after 20 seconds. This process needs 20 seconds of CPU time to complete its execution. It performs some I/O and sleeps 1.89 seconds over its life time. User B executes two long run processes with a ticket ratio of 3 to 2. The length of the accounting window attached to each process for CPU usage accounting is two seconds. Figure 11 shows the allocated share for each process measured over one second intervals.

When process 1 starts execution after 20 seconds, active tickets are modified so that user B processes' defined shares become 30 and 20% respectively. Since process 1 suspends its execution by itself, it is boosted on wake up and then penalized if necessary. In such a manner, fairness is improved compared to a strict fair-share scheduling.

If the slept time of process 1 waiting for I/O completion is exactly compensated with the boost on wake up, process 1 can complete its execution in 40 seconds. This execution time may varies with the accuracy of the compensation, however. The **Table 2** shows execution time of process 1 with various window length. CPU time allocation error slightly increases with the window length. This is due to a less stable distribution of CPU time among processes. Indeed, as the window gets longer, it takes more time for a process to be allocated its defined share over the time interval of the window. The resulting boosts are thus highly enhanced.

Table 2 Total execution time of process 1 of the example of Fig. 11 with various window size.

Window length	Execution time	Error
1	40.09	−0.2%
2	39.46	1.35%
4	39.209	1.98%
10	38.68	3.3%

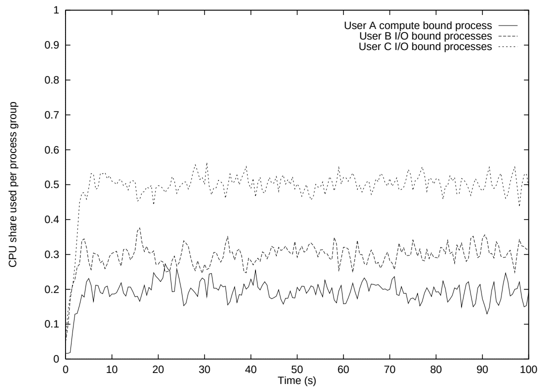

Fig. 12 User A executes a single compute bound process, users B and C run the same load of I/O bound processes.

4.2.3 I/O Bound Processes

In this experiment, user A runs a single compute bound job, user B and C both executes five I/O bound processes which repeatedly run and sleep for 20 ms and 80 ms in the average respectively. The defined shares are 20%, 30% and 50% for user A, B and C. **Figure 12** shows the measurement over 4 seconds intervals of the allocated share to each users. The accounting window time length is 2 seconds.

During this simulation for 100 sec, allocated CPU times of user A, B, and C are 50.338 sec, 30.223 sec and 19.439 sec, respectively. It means that the CPU time allocation errors are 0.6%, 0.7% and −2.8% for user A, B, and C. Even in the case of a high interactive load, the scheduler allows to precisely allocate CPU time to each user.

5. Conclusion

In this paper, we presented a fair-share scheduling method which combines a classical priority scheduler with stride scheduling. Using priority scheduling improves both fairness and the response time of interactive and I/O intensive applications by boosting the execution of processes on wake up.

As processes are scheduled using priority scheduling on wake up until they are allocated their fair share, all processes not using all their allocated CPU time will always be scheduled ahead of compute bound applications. In particular, in lightly loaded systems, all users requests can be satisfiable without fair-share allocation of processing time using the implemented classical scheduler. On the contrary, if the load increases, our implementation of stride scheduling allows to allocate processing time fairly among users and processes. The precise control over CPU time allocation is obtained using a short time sweeping window recording recent CPU usage of processes, allowing to keep the scheduling algorithm simple. Experiments have shown that fairness is obtained with a small error, even in the case of high interactive loads.

The event based update of priorities and tickets allocation also allows to reduce the scheduling overhead and combined with an improved implementation of stride scheduling. Our implementation results on our emulator have shown that the total overhead do not depend on the number of runnable processes. However, an implementation on a real system is desired to validate the implementation choices more precisely using a more realistic load.

Acknowledgments We would like to thank Mentor Graphics Japan Corporation for providing their products and services as a part of the Higher Education Program.

A part of this research was supported by the Grant-in-Aid for Scientific Research (B)(2)#12558027, and #13480083 from Japan Society for the Promotion of Science and also by the Grant-in-Aid for Scientific Research on Priority Areas (C)#13224050 from the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

References

1) Anglano, C.: A Fair and Effective Scheduling Strategy for Workstation Clusters, *Proc. Int'l Conf. on Cluster Computing* (2000).
2) Arpaci-Dusseau, A.C. and Culler, D.E.: Extending Proportional-Share Scheduling to a Network of Workstations, *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications* (1997).
3) Chandra, A., Adler, M., Goyal, P. and Shenoy, P.: Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors, *Proc. 4th USNIX Symp. on Operating Systems Design and Implementation* (2001). http://www.usenix.org/
4) Demers, A., Keshav, S. and Shenker, S.: Analysis and Simulation of a Fair Queueing Algorithm, *Proc. ACM SIGCOMM*, pp.1–12, (1989).
5) Duda, K.J. and Cheriton, D.R.: Borrowed-Virtual-Time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler, *Proc. 17th ACM Symp. on Operating Systems Principles*, published as *Operating System Review*, Vol.34, No.5, pp.261–276, (1999).
6) Essick, R.B.: An Event-based Fair Share Scheduler, *Proc. Winter 1990 USENIX Conf.*, pp.147–161 (1990).
7) Hwang, K. and Jin, H.: Designing SSI Clusters with Hierarchical Checkpointing and Single I/O space, *IEEE Concurrency*, pp.60–69, January-March Issue (1999).
8) Kay, J. and Lauder, P.: A Fair Share Scheduler, *Comm. ACM*, Vol.31, No.1, pp.44–55 (1988).
9) Khanna, S., Sebree, M. and Zolnowsky, J.: Real-time Scheduling in SunOS 5.0, *Proc. Winter 1990 USENIX Conf.* (1992).
10) Le Moal, D., et al.: A Fair Share Priority Scheduler for a Distributed Operating System, *IPSJ SIG Notes* (SWoPP'99), 99-OS-82, pp.49–56 (1999).
11) McKusick, M.K.: *The Design and Implementation of the 4.4BSD Unix Operating System*, Addison Wesley (1996).
12) Nieh, J. and Lam, M.: The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Application, *Proc. 16th ACM Symp. on Operating Systems Principles*, pp.184–197 (1997).
13) Nieh, J., Vaill, C. and Zhong, H.: Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler, *Proc. 2001 USENIX Annual Technical Conference*, pp.1–15 (2001). http://www.usenix.org/
14) Petrou, D., Mildford, J.W. and Gibson, G.A.: Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers, *Proc. 1999 Usenix Annual Technical Conference*, pp.1–14 (1999). http://www.usenix.org/
15) Andleigh, P.K.: *Unix System Architecture*, Prentice Hall (1990).
16) Waldspurger, C.A. and Weihl, W.E.: Lottery Scheduling — Flexible Proportional-Share Resource Management, *Proc. First Symp. on Operating Systems Design and Implementation*, pp.1–11 (1994).
17) Waldspurger, C.A. and Weihl, W.E.: Stride

Scheduling — Deterministic Proportional-Share Resource Management, Technical memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science (1995).
18) Wang, S., Wang, Y-C. and Lin, K-J.: A Priority-Based Weighted Fair Queueing Scheduler for Real-Time Network, *Proc. Int'l Conf. on Real-Time Computing Systems and Applications* (1999).
19) System services and Application Packaging tools, AT&T Unix System V Release 4 Programmers Guide, Prentice Hall (1990).

(Received January 24, 2002)
(Accepted May 23, 2002)

Damien, Le Moal was born in 1972. He received an engineering degree in Computer Science from the National Engineering School of Electronics, Electrotechnics, Hydraulics and Computer Science of Toulouse (ENSEEIHT, France) in 1995 and M.E. in computer science from Kyoto University (Japan) in 2000. Since 2000, he is working at Hitachi Ltd. Systems Development Laboratory (Japan). His research interests include operating systems, real-time systems and communication.

Masahiro Ikumo is currently a graduate student of Graduate School of Informatics, Kyoto University. His reserch interests include process scheduling in operating system and scientific visualization. He received the B.E. degree in Information Science from Kyoto University in 2001.

Tomoaki Tsumura was born in 1973. He received his M.E. degree from Kyoto University in 1998. Since 2001 he has been an research associate in Graduate School of Economics, Kyoto University. His research interests include computer architecture for brain-like computing and parallel processing. He is a member of IPSJ and JSAI.

Masahiro Goshima was born in 1968. He received his M.E. degree from Kyoto University in 1994. He was a research fellow of the Japan Society for the Promotion of Science from 1994. Since 1996 he has been in Kyoto University as an assistant professor. He has been engaging in the research area of high-performance computing systems. He received IPSJ Yamashita SIG research award and IPSJ best paper award in 2001 and 2002, respectively. He is a member of IPSJ and IEEE.

Shin-ichiro Mori was born in 1963. He received his B.E. in electronics from Kumamoto University in 1987 and M.E. and Ph.D. in computer science from Kyushu University in 1989 and 1995, respectively. From 1992 to 1995, he was a research associate in the Faculty of Engineering, Kyoto University. Since 1995, he has been an associate professor in the Department of Computer Science, Kyoto University. His research interests include computer architecture, parallel processing and visualization. He is a member of IEEE, ACM, EUROGRAPHICS, IEICE and IPSJ.

Yasuhiko Nakashima received the B.E., M.E. and Ph.D. degrees in Computers Engineering from Kyoto University, Japan in 1986, 1988, and 1998, respectively. He was a computer architect at Computer and System Architecture Department, FUJITSU Limited in 1988–1999. Since 1999 he has been an Associate Professor in Graduate School of Economics, Kyoto University. His research interests include processor architecture, emulation, CMOS circuit design, and evolutionary computation. He is a member of IEEE CS, ACM and IPSJ.

Toshiaki Kitamura received the B.E., M.E. and Ph.D. degrees in Computers Engineering from Kyoto University, Japan in 1978, 1980, and 1996, respectively. He is currently a Professor on the faculty of Information Sciences, Hiroshima City University. Previously, he was a computer architect at Computer and System Architecture Department, FUJITSU Limited in 1983–2000. From 2000 to 2002, he was an Associate Professor in Center for Information and Multimedia Studies, Kyoto University. His research interests include processor architecture, emulation, CMOS circuit design, and evolutionary computation. He is a member of IEEE, ACM, IEICE and IPSJ.

Shinji Tomita was born in 1945. He received the B.E., M.E. and Ph.D. degree in Electronics from Kyoto University in 1968, 1970, and 1973, respectively. He was a research associate from 1973 to 1978 and an associate professor from 1978 to 1986 both in the Department of Computer Science, Kyoto University. From 1986 to 1991, he was a professor in the Department of Information Systems, Kyushu University. From 1991 to 1998, he was a professor in the Faculty of Engineering, Kyoto University. Since 1998, he has been a professor in the Graduate School of Informatics, Kyoto University. His current interests include computer architecture and parallel processing. He is a member of IEEE, ACM, IEICE and IPSJ. He was a board member of IPSJ directors in 1995, 1996, 1998 and 1999. He is a fellow of IEICE and IPSJ.