PRIORITY ENHANCED STRIDE SCHEDULING

Damien Le Moal[†]*, Mineyoshi Masuda, Masahiro Goshima,*
Shin-ichiro Mori, Yasuhiko Nakashima,
Toshiaki Kitamura and Shinji Tomita

Graduate School of Informatics, Kyoto University
Yoshida-hon-machi, Sakyo-ku, Kyoto, 606-8501, JAPAN

(†:Current affiliation is Systems Development Lab., Hitachi Ltd., Japan)

ABSTRACT

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 interactive processes execution is not degraded. It is also shown that the scheduling overhead is bounded and does not depend on the number of runnable processes.

[Keywords]

Distributed Operating System, Process Scheduling, Fair-share scheduling, Priority Scheduling, Stride Scheduling, Responsiveness

1 INTRODUCTION

Workstations are still commonly used as desktop machines allowing to share coarsely resources distributed over a local area network. Operating systems used in such environment, in particular UNIX like systems, are thus mainly concerned with executing applications of a single user logged at the console. Users generally expect from such systems to be responsive to their input. As a result, classical scheduling methods, like priority scheduling are commonly used to favor interactive and I/O bound processes while maximizing processor use for compute bound applications.

However, development of low-latency communication mechanisms combined with the use of more efficient execution models have allowed to change the definition of such environment to more tightly coupled machines getting closer to massively parallel processors. In particular clusters of workstations are actively developed providing a low-cost and incrementally-scalable environment supporting finer parallelism in application execution. This evolution potentially changes the load executed on classical desktop workstations from sequential processes of a single user to a variety of applications run by a large number of users.

In such environment, it is desirable to guarantee that each user can be allocated fairly resources and in particular processing time. Most scheduling policies permit a user running many processes to receive more processor time at the expense of users running only few processes. This problem can be exacerbated in a distributed environment to the point where a single user is able to use all resources throughout the network.

To schedule fairly applications in a cluster, one should first be able to control precisely processor time allocation within a single processing element. The local scheduler should allocate CPU time among processes so that one user cannot degrade another user process execution by running more processes. Interactive and I/O bound application should also be guaranteed a good response time so that users are not disappointed by the evolution of their computing environment. Applications which can be compute bound, interactive or I/O intensive that may each be serial or parallel should be all scheduled efficiently while being fair. Fairness and responsiveness are both needed.

Recent research works have produced several specialized scheduling method implementing a fair-share allocation of processing time like stride-scheduling. In a multi-user environment, all users can receive the same share of CPU time regardless of the number of processes they execute. However, such methods only performs 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 briefly priority and stride scheduling. Section 3 presents in more detail the scheduling algorithm used and its implementation. Section 4 shows some experiment results, finally section 5 concludes this paper.

2 BACKGROUND

This section first discusses priority scheduling and presents stride scheduling. A work combining priority scheduling with another fair-share CPU time allocation method is also discussed.

2.1 Priority Scheduling

Priority scheduling is widely used in conventional operating systems because it allows to efficiently executes interactive processes by emulating the short-job-first policy using dynamically calculated priorities. However, users running several processes can receive more CPU time at the expense of other users running only few processes.

Fair-share schedulers can also be built on top of such classical priority schedulers(Essick, 1990)(Key and Lauder, 1988). Such schedulers regularly recompute all processes priority depending on the past CPU usage and on the share of CPU time allocated to users or processes. The work presented in 'Key and Lauder(1988)' recomputes all process priorities every 4 seconds, with fairness realized over hours or days. The work presented in 'Essick(1990)' provide better results for fairness, but it is not precise because considering a lot of parameters to penalize or boost processes execution depending on their CPU usage.

2.2 Stride Scheduling

Stride scheduling(Waldspurger and Weilhl, 1995) is a deterministic fair-share CPU scheduling method. It uses a currency called "ticket" to encapsulate CPU share allocation rights. A process with t tickets in a system with a total of T tickets is allocated $\frac{t}{T}$ of the CPU time. A hierarchical allocation of tickets to users and processes allows to both define the share of a user and of his processes. Moreover, by only considering active tickets, which are tickets of runnable processes, the defined share calculation can generates simply the maximal possible share depending on the current load of the system.

Stride scheduling allocates a constant time quantum to each process at regular intervals. The basic idea is to compute the time (i.e. stride) a process must wait before receiving its next time quantum. The stride of a process is thus inversely proportional to its defined share. A "pass" parameter is associated to each process: each time a process is scheduled, its pass is incremented by its share. The scheduler always chooses the process with the smallest pass for execution.

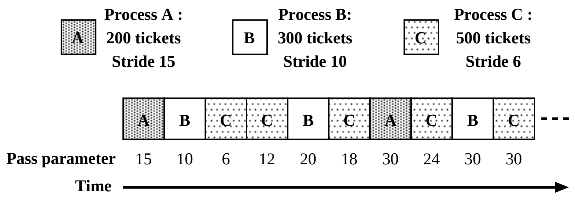
A
Process A : 200 tickets Stride 15
B
Process B: 300 tickets Stride 10
C
Process C : 500 tickets Stride 6
A B C C B C A C B C
Pass parameter 15 10 6 12 20 18 30 24 30 30
Time

Figure 1: *Three processes with an initial pass of 0 are executed using stride scheduling. The process with the minimum pass is scheduled at each interval.*

Figure 1 shows an example where three processes with a ticket allocation ratio of 2:3:5 are scheduled with stride scheduling. The initial pass is 0 for each process, and are first scheduled in the order A, B and C. At the end of the sequence represented, all processes pass are equal, the same pattern will then be repeated again. Stride scheduling thus allows to achieve fairness over very short time intervals. However, like other strict fair-share scheduling methods, for example lottery scheduling(Waldspurger and Weihl, 1994), it suffers from several problems.

- A search or sort of the ready queue is necessary so that processes can be selected in increasing pass order, resulting in a cost of at least $O(\log n)$ for queue manipulation, where n is the number of runnable processes.
- The run time behavior of processes is not considered, which results in the following two problems.
  - The dispatch latency on wake up depends highly on the share of CPU a process was allocated because processes waking up after waiting for I/O completion do not preempt the currently running process.
  - A process sleeping waiting for I/O completion cannot be allocated its fair share as the time slept is lost and cannot be simply compensated.

Some improvement of the basic algorithm(Waldspurger and Weilhl, 1995)(Arpaci-Dusseau and Culler, 1997) can provide better results for interactive process execution by boosting or penalizing processes depending on their behavior. But this necessitate a complex dynamic ticket allocation mechanism which increases the total overhead.

2.3 Hybrid lottery scheduling

The work presented in 'Petrou, Mildford and Gibson(1999)' solves the problems of strict fair-share scheduling by combining the system level priorities of the FreeBSD scheduler with the fair-share scheduling method called lottery scheduling(Waldspurger and Weihl, 1994). 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.

The time slept doing I/O is compensated with a ticket allocation boost on wake up so that processes can be scheduled more often when returning to user mode. Lottery scheduling was also modified to support process preemption, but this modification needs also a dynamic ticket adjustment to preserve the overall fairness.

While improving the responsiveness of interactive applications compared to the basic lottery scheduling, this method increases the amount of computation necessary at each schedule operation. Scheduling overhead also increased and the implementation is difficult because of the complexity of the necessary adjustment to not degrade the fair allocation of CPU time.

3 ENHANCED STRIDE SCHEDULING

Extending fair-share scheduling using priority scheduling can provide the desired control over processes execution order while allocating fairly 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 passes. 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 figure 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.

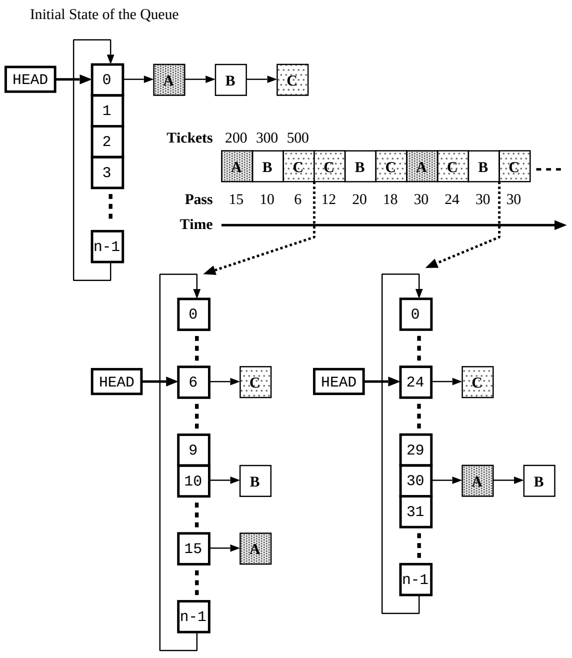

Figure 2: *Example of figure 1 using our implementation of stride scheduling.*

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 previous list index of this process (the index of the head list where it was chosen) plus 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 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.

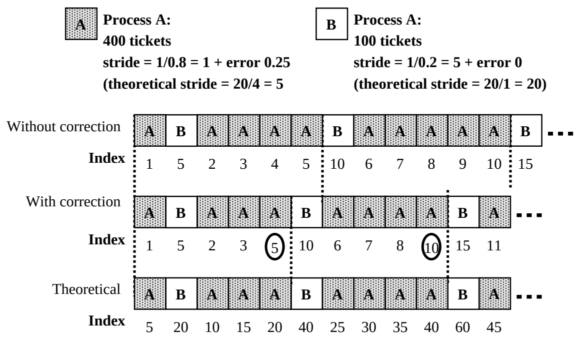
A
Process A:
400 tickets
stride = 1/0.8 = 1 + error 0.25
(theoretical stride = 20/4 = 5
B
Process A:
100 tickets
stride = 1/0.2 = 5 + error 0
(theoretical stride = 20/1 = 20)
Without correction
A B A A A A B A A A A A B ...
Index
1 5 2 3 4 5 10 6 7 8 9 10 15
With correction
A B A A A B A A A A B A ...
Index
1 5 2 3 5 10 6 7 8 10 15 11
Theoretical
A B A A A B A A A A B A ...
Index
5 20 10 15 20 40 25 30 35 40 60 45

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

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 figure 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 1024 so that the possible minimal share is equal to $1/1024 \approx 0.001 = 0.1\%$.

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. 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, and scheduling should be preemptive to minimize the wait time in ready queue on wake up.

As the CPU time allocated may be higher than the defined share of a process on a certain interval of time, depending on the past CPU usage, basically two cases can 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 such case, the boost resulting by the over share allocation should be compensated with a "*unboost*" to preserve the overall fairness, so that processes which used the cpu cycles unused during they were sleeping are not penalized.

To be reactive to such situations, the scheduler should not only consider the total CPU time allocation over the life time of a process, but also should measure the recently allocated share over shorter intervals. A accounting window recording the past CPU allocation of a process can be used. 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.

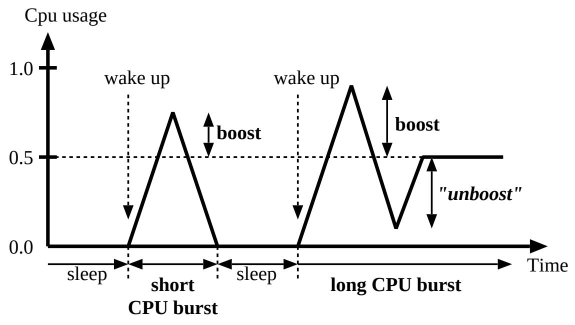

Figure 4: *I/O bound processes must be boosted on wake up. When having a longer CPU burst, the boost must be compensated to preserve the overall fairness.*

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 so that others 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 implementing this policy.

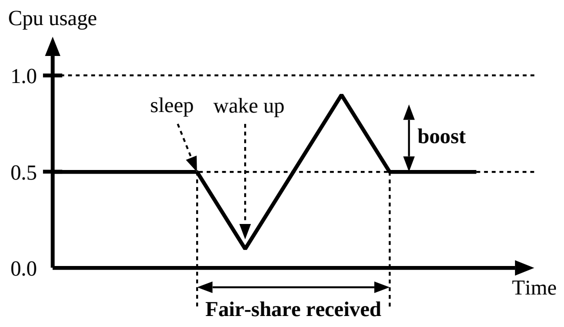

Figure 5: *A process having a long CPU burst and sleeping only very short intervals can still be allocated its defined share.*

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 to control precisely the "*unboost*" of a process.

The ready queue has a classical multilevel feedback queue structure, as shown in figure 6. The stride scheduling queue presented in figure 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.

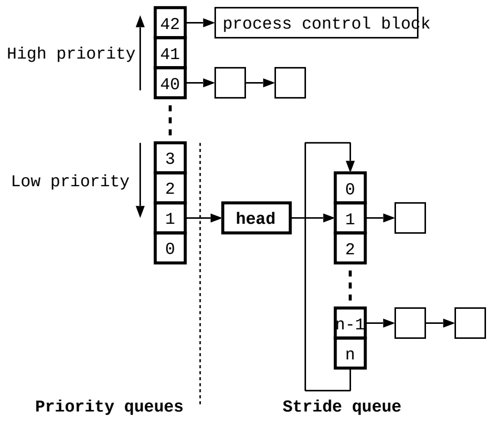

Figure 6: *The ready queue is a classical multilevel feedback queue structure using priority scheduling between queues. The stride queue is assigned a fixed priority (priority 1).*

Figure 7 shows the table used as a base to calculate priorities and time quantums. For each priority level, this table defines the next priority `Pexp` when a process uses all its time quantum, the priority on wake up `Pslp` if a process sleeps waiting for an event from this priority level and 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

```
int pritbl[] = {
        // PRIO     TQ      Pexp     Pslp
        /*  0 */    100,     0,       0,      // Kernel threads
        /*  1 */    100,     1,       32,     // Stride queue
        /*  2 */    100,     2,       33,     // normal priority
        /*  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
```

Figure 7: *Priority table.*

processes tend to occupy more priority levels.

High priority processes are assigned a short time quantum (for example 20ms) when scheduled. This time quantum is longer for lower priority levels and is set to 100ms 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 provably 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 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 0.
- On the contrary, if the preempted process was allocated its fair-share or more, its priority is set to 0 so that it will be scheduled next time using stride scheduling.

A process will thus be scheduled ahead of processes in the stride queue using priority scheduling until it is allocated its fair-share, resulting in the desired boost. As a process can then receive in a row the processing time 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

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.

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

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

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 always releasing the CPU before the end of their time quantum may be able to use more than their defined share. As a solution, a control point is added on return from system call, when 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 from boosting several times in a short interval a process always sleeping just at the end of its boost while being scheduled with its priority.

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 scheduling overhead measurements are shown, some experiments results are discussed next.

4.1 Scheduling Overhead

We have compared the relative overhead of stride scheduling queue manipulation for our implementation and for the implementation proposed originally(Waldspurger and Weilhl, 1995). Figure 8 shows the cost of put and get operations for both implementations depending on the number of runnable processes. Those measurements were done on a 300MHz 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.

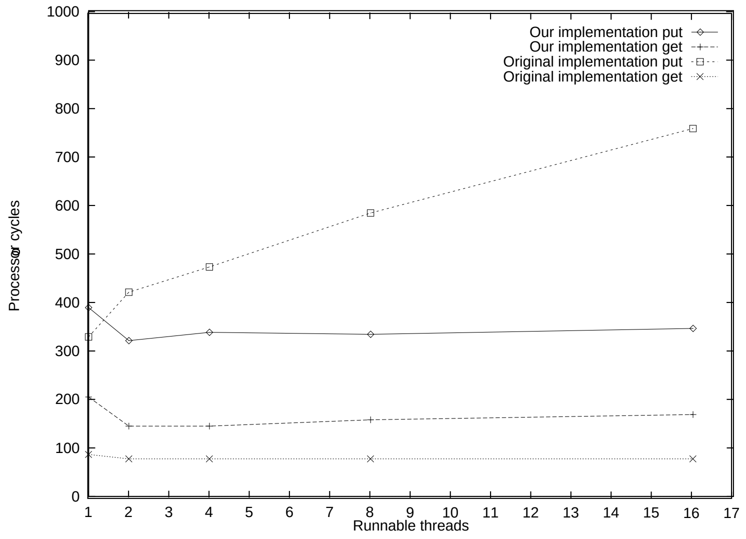

Figure 8: *Stride queue manipulation cost (put and get operations).*

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 its priority and put it back into the ready queue if 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 when only priority scheduling is used or only stride scheduling. 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 this two limits as 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.

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.

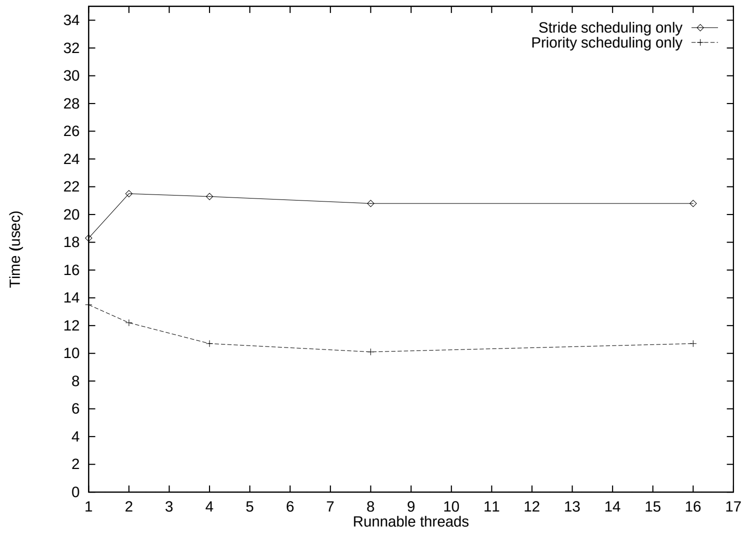

Figure 9: *Scheduling overhead when either priority scheduling or stride scheduling is used.*

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 respective length 300 milliseconds, 700 milliseconds and 5 seconds. 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.

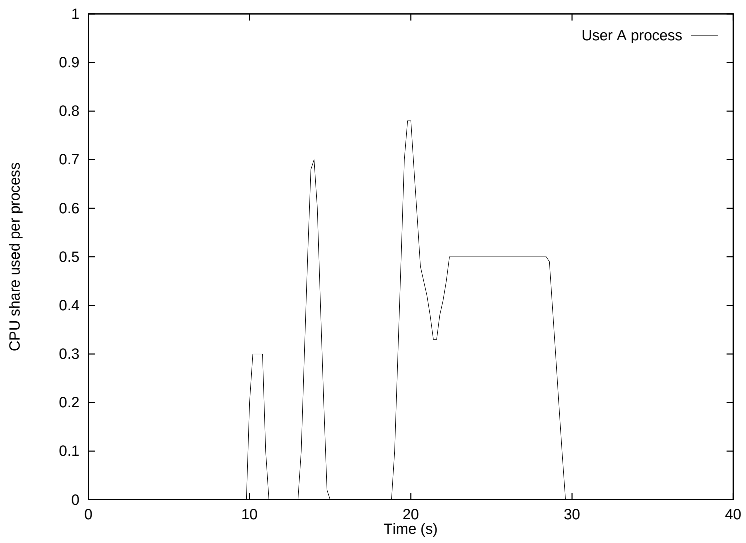

Figure 10: *Basic scheduling behavior for different CPU bursts length.*

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, and to preserve fairness user A process is penalized. 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 figure 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, but this graph shows the allocated share for each process measured over one second intervals.

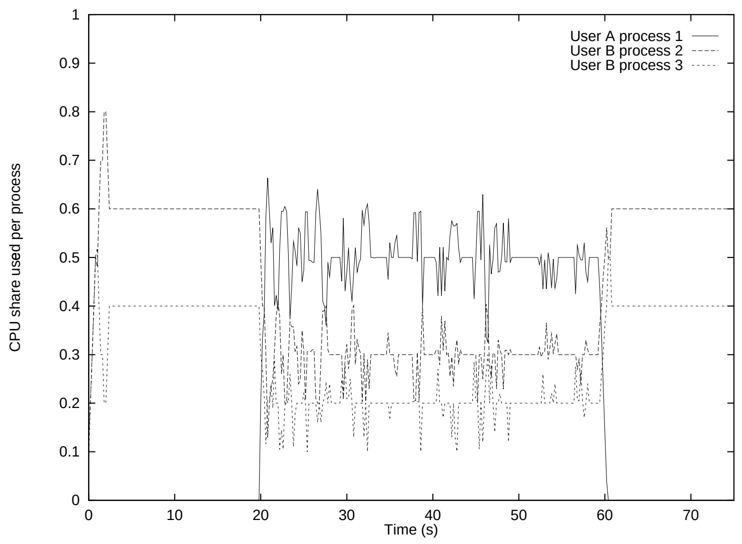

Figure 11: *Two users A and B with the same defined share run compute bound processes. User A process starts after 20 seconds.*

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.

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%

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

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 of higher amplitude.

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 in the average run for 20ms and sleep for 80ms. The defined shares are respectively 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.

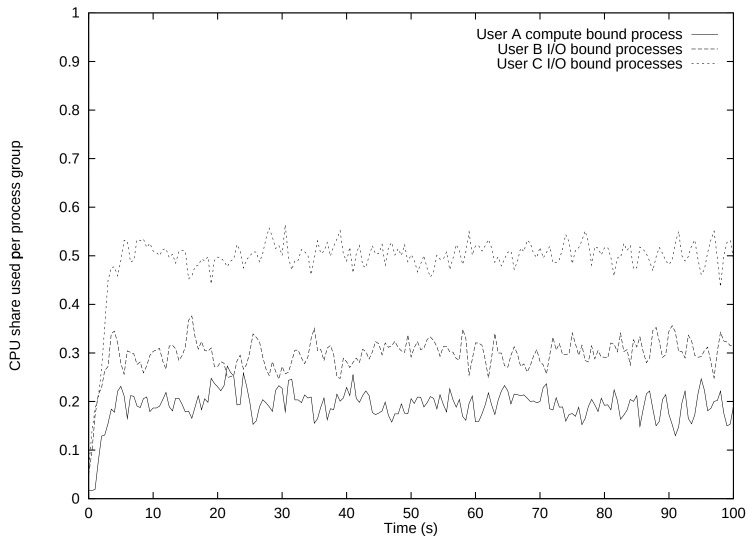

Figure 12: *User A executes a single compute bound process, users B and C run the same load of I/O bound processes.*

Over the length of this simulation, user A was allocated 50.338 s of the CPU time, user B 30.223 sec and user C 19.439 sec, resulting in respective errors of 0.6%, 0.7% and -2.8%. 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 fairly processing time 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 a high interactive load.

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, measurements presented have shown that the total overhead do not depend on the number of runnable processes. However, an implementation on an exploitation system is desired to validate the implementation choices more precisely using a more realistic load.

ACKNOWLEDGMENT

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 the Ministry of Education, Science, Sports and Culture of Japan.

REFERENCES

McKusick,M.K.(1996): The Design and Implementation of the 4.4BSD Unix Operating System. Addison Wesley.

Essick,R.B.(1990): An Event-based Fair Share Scheduler. USENIX Conference Proceedings (Winter 1990).

Kay,J. and Lauder,P.(1988): A Fair Share Scheduler. Communications of the ACM, Volume 31, Number 1.

Waldspurger,C.A., Weihl,W.E.(1994): Lottery Scheduling - Flexible Proportional-Share Resource Management. Proc. of the first Symp. on Operating Systems Design and Implementation.

Petrou,D., Mildford,J.W., Gibson,G.A.(1999): Implementing Lottery Scheduling : Matching the Specializations in Traditional Schedulers. Usenix Annual Technical Conference, http://www.usenix.org, June 1999.

Waldspurger,C.A., Weihl,W.E.(1995): Stride Scheduling - Deterministic Proportional-Share Resource Management. Technical memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

Arpaci-Dusseau,A.C., Culler,D.E.(1997): Extending Proportional-Share Scheduling to a Network of Workstations. Proc. of Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA).