分散システムにおける Fair Share プライオリティ・スケジューラ

Damien Le Moal[†] 鶴崎剛大[†] 増田峰義[†]
伊達新哉[†] 山添博史[†] 五島正裕[†]
森眞一郎[†] 富田眞治[†]

本論文では Stride Scheduling という Fair-Share スケジューリング手法の新しい実装とその方法のプライオリティスケジューリング法との組み合わせを提案する。この Fair-Share プライオリティ・スケジューラではユーザとプロセスに CPU 時間を割り当てることが可能になり、またプロセスの実行順序を制御出来るので、インタラクティブなプロセスの応答時間を短くすることが出来る。このスケジューリング方法を評価した結果、短い時間間隔で効率的に CPU バウンドプロセスの公平なスケジューリングを実現出来ると分かった。最後にこの手法のインタラクティブプロセスに対応するための拡張を述べる。

A Fair Share Priority Scheduler for a Distributed Operating System

DAMIEN LE MOAL,[†] TAKEHIRO TSURUSAKI ,[†] MINEYOSHI MASUDA,[†]
SHIN-YA DATE,[†] HIROSHI YAMAZOE,[†] MASAHIRO GOSHIMA,[†]
SHIN-ICHIRO MORI[†] and SHINJI TOMITA[†]

In this paper, after discussing some commonly accepted scheduling methods, we present our novel implementation of stride scheduling, a fair-share scheduling scheme. To allow a flexible and controllable allocation of cpu time among users and processes while preserving a good response time for interactive and time critical jobs, this method is combined with a classical priority scheduling scheme. Evaluation results show that fairness can be ensured very efficiently within very short time intervals for compute bound jobs, even in the case of dynamic adjustment of allocated share. Finally, we discusses some extensions of this scheme to handle efficiently interactive processes.

Keywords : Distributed Operating Systems, Process Scheduling, Fair-share.

1. Introduction

Computer Colony system is a cluster of computers, actually developed in our laboratory. Colonia, the distributed operating system of Computer Colony, implement a uniform system image, so that each node can be seen as a processing element of a distributed shared memory (massively) parallel computer.

In such environment, we assume that several users must be able to concurrently execute various applications ranging from some light interactive processes such as a simple terminal, to heavy background parallel jobs.

Considering such characteristics, the following two points are of major importance.

Fairness Unlike Unix systems, logging in Colonia can give a user access to the whole system computing power. To prevent this user from getting too much resources at the expense of others, a user should not be able to receive more than the share of cpu time it was allocated by the system, and this should be ensured within a short period of time.

Response Time It is reasonable to think that compute bound jobs may be allocated a high share of cpu time so that they can complete execution fastly. Interactive processes should not suffer from running concurrently in such environment, and should be scheduled quickly after waking up, in order to reduce their response time to event occurrence.

The process scheduling scheme used on such sys-

---

† 京都大学 情報学研究科
Graduate School of Informatics, Kyoto Univ.

tem must then ensure both of these characteristics : achievement of fairness within a short time interval, and a small response time. Moreover, this is not only necessary for a single node, but also system wide, particularly fairness. Our approach is to develop a scheduling scheme for the node level realizing both fairness and a good response time, and to extend it to the system wide level.

In this paper, we only consider process scheduling over a single node of the system. The next section presents some current scheduling methods, in particular stride scheduling. Section 3 discusses our scheduling scheme which combine both priority and stride scheduling, some evaluation results are shown in section 4. Finally, section 5 will conclude this paper.

2. Background

Conventional operating systems commonly use the notion of priority to control precedence of process execution to ensure a good response time, without consideration for fairness. On the other hand, several low-overhead, efficient, fair share scheduling methods where developed [9,10], but do not allow a precise control of process execution order.

2.1 Classic Priority scheduling

This scheduling method is widely used because of its simplicity, ensuring a low overhead and a maximal utilization of the processor. Priorities can be calculated in several ways, and classically, processes sleeping often waiting for events are given a high priority on wake up, ensuring a good response time to event occurrence, thus increasing process throughput and user productivity. But this scheme suffers from several drawbacks. Mainly, it is difficult to tune and in its basic implementation, priority scheduling does not allow to schedule fairly processes over time. As a result, users with many processes can easily receive more processing time. Also, a costly regular adjustment of process priority is necessary to avoid cpu starvation.

2.2 Priority-Based Fair share scheduling

Considering the previous problems, some priority based fair share schedulers have been developed [12]. By measuring the past utilization of processor time for each process, and comparing it to the allocated share, priority are adjusted to give more cpu time to those processes which received less than their share, and to lower the cpu use of those which received too much. Good results can be achieved, but these methods are difficult to tune and highly undeterministic. A long time interval to ensure a fair allocation of cpu time is also needed (for some implementation, in the order of minutes or more).

2.3 Stride Scheduling

Stride scheduling is a deterministic resource allocation algorithm for time sharing systems which can be used as a process scheduling method to allow a precise control over cpu time allocation. It was first presented in [9] and can be seen as an application of rate-based network flow control algorithms to process scheduling.

2.3.1 Share and Tickets

In basic stride scheduling, resource allocation is determined using an amount of tickets used as a currency. Basically, a process with t tickets in a system with T tickets is allocated t/T share of cpu time. Therefore, the more a process owns tickets, the more it can receive processing time.

Some variations of this basic allocation method use exhausting tickets (tickets whom validity is timely limited) with either a loan and borrow mechanism or system credits [11] to handle more efficiently interactive processes fairness. With these extensions, runnable processes can borrow the tickets of the sleeping one to increase their share, thus their throughput. But they must pay back the lent process when it wakes up, increasing its share and its chances to be scheduled quickly. These methods cannot anyway efficiently handle a heavy interactive load or processes with very short run time. The overhead introduced is also not negligible.

2.3.2 Basic algorithm

Classically, in time sharing systems, processes are regularly allocated a time interval (time slice or time quantum). If a process consumes up its allocated time slice, the system preempts it and allocate the next time slice to another process if one is available. In stride scheduling, the core idea is to compute the time interval, or stride a process must wait before receiving its next time quantum. The stride of a process is inversely proportional to its share. Each time a process is scheduled, a pass parameter associated to this process is incremented by the stride of the process. At each time quantum, the process with the lowest pass is chosen.

Figure 1 shows an example of stride scheduling where three processes compete for the cpu with different ticket allocation. Process 1 has a stride of 6 with 500 tickets, process 2 a stride of 10 with 300 tickets and process 3 a stride of 15 with 200

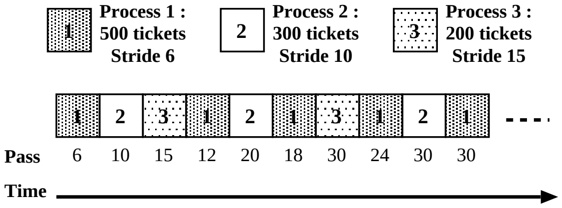

Fig.1 *Three runnable processes with different allocation of cpu share are scheduled using stride scheduling : the process with the smallest pass is scheduled first.*

tickets. Considering the ticket ratio of each process, process 1 must receive half of the cpu time $(500/(500+300+200))$, process 2 and 3 sharing the other half in the ratio of 3 to 2. At the initial state, all processes pass are 0, processes are first scheduled in the 1,2,3 order. Over the scheduling sequence of the example, process 1 is scheduled 5 times, process 2 and 3 three and two times respectively. At the end of this sequence, all processes pass are equal. The represented pattern will then be repeated over and over. With a hundred milliseconds time slice, as the length of this pattern is ten, fairness is achieved over one second intervals.

2.3.3 Original implementation

The first implementation, proposed in [9], used a sorted queue as the data structure to manage processes, with a cost order of $O(\log n)$ for queue manipulation, where n is the number of processes in the queue.

Using a floating point for the stride is possible, but for simplicity, an integer type was used. The stride of each process is then calculated as the ratio of a constant to the share of the process, and rounded up to the nearest integer. The constant used is chosen big enough (2^{20}), so that the round error becomes very small and practically undetectable.

As the stride is calculated as a big integer, the pass parameter may grow quickly. To avoid a fast overflow, a 64 bit integer was used so that the time until this parameter overflow is long enough to not be considered practically as a problem. In the case of a 32 bit integer use, the pass parameter can overflow more frequently, thus needing periodical adjustment of all processes pass parameter.

3. Fair Share Priority Scheduler

This section presents in detail our implementation of stride scheduling, as well as its combination with a simple priority scheduling method.

3.1 Stride scheduling implementation

This section presents in detail our implementation of stride scheduling, as well as its combination with a simple priority scheduling method.

3.1.1 Data Structure

In its basic implementation, stride scheduling has two main drawbacks : each process pass parameter may overflow after a certain time as it is incremented by the process stride each time the process is scheduled. Also a costly search for the process with the smallest pass is necessary at each schedule operation. Our implementation uses the basic algorithm presented in the previous section, but avoids its problems by using a circular array of process record linked lists, as shown in figure 2.

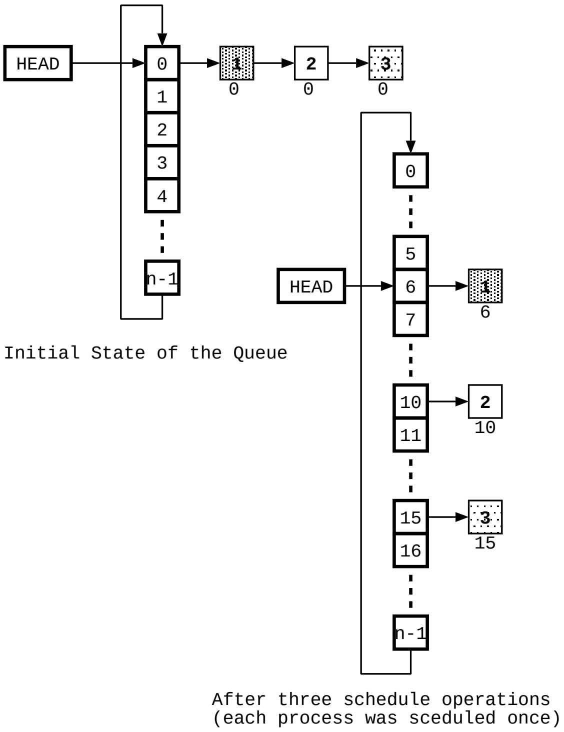

Fig.2 *Stride queue data structure : an array of list of process record. The initial state of the queue for the example of figure 1 is represented, as well as the state after three schedule operations.*

Each list contains processes equal pass, these processes are ordered in first in first out (as they all have the same pass, no distinction can be made). The head list contains all processes with the smallest pass. All get operations on this queue are thus done in this list, the head pointer is advanced to the next non empty list each time the head list becomes empty.

If a process uses up its time slice without sleeping, it is put back into the list at an array index equal to its previous index (the index of the head list) plus the current stride of this process, thus virtually incrementing this process pass by its stride. Doing so, this method also avoids ordering a queue with the pass parameter, as lists are naturally ordered in increasing pass form the list pointed to by the head.

Manipulating the stride queue do not depend any more on the number of processes, resulting in a bounded time for all put and get operations on this structure. Incrementing the pass parameter is also no longer necessary, avoiding this parameter overflow and the resulting necessary corrections.

3.1.2 Ticket Allocation and Share Calculation

We consider two different sorts of tickets : user tickets and process tickets. User tickets are allocated to users by the system at login time, and are said to be active if the user owning these tickets has at least one process runnable or running in the node. An active user i owning tu_i active tickets in a system with $Tu = \sum tu_j$ active user tickets is allocated $\frac{tu_i}{Tu}$ of the computing time. Each user allocate process tickets to his processes, those tickets are said to be active if the process owning them is running or in runnable state.

The share of cpu time accessible to a process is then calculated with

$$Process\ k\ share = \frac{tu_i}{\sum_j tu_j} * \frac{tp_{i,k}}{\sum_j tp_{i,j}} \qquad (1)$$

where $tp_{i,k}$ is the number of tickets allocated to the process k of user i and $\sum tp_{i,j}$ the sum of active process tickets for the user i.

Active tickets sums are maintained on event occurrence which changes any process state : each time a process becomes runnable or goes to sleep, the sum of active process tickets of the user owning this process is updated and if necessary the sum of all user active tickets is also adjusted.

As a user may not know how many processes he is running (which is a reasonable assumption : consider the number of processes started when a user logs in a system) no limit is put to the total number of tickets allocated to processes by the user owning them. Cpu time is always shared among all runnable processes depending on the relative ratio of their tickets (figure 3)

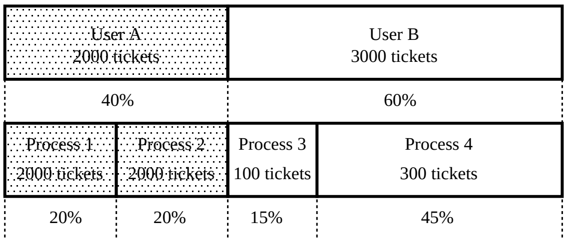

Fig.3 *An example of ticket allocation with the resulting cpu share for each process.*

3.1.3 Basic Stride Calculation

Using a floating point for the stride value is possible, but considering our data structure, the stride must be an integer. As the stride must be inversely proportional to the share, a division is needed, and the result must be rounded up to an integer value. The stride of a unit is calculated using the formula:

$$Process\ stride = rint(\frac{\lambda}{process\ share}) \qquad (2)$$

Where `rint()` is the function rounding a floating point to the nearest integer. A correct (perfect) calculation of the stride with no round error would use λ equal to the lowest common multiple of all process share fraction numerator, which is too costly to be calculated at each scheduling operation. To avoid this problem, λ is chosen as a constant. However, as the size of the stride queue 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. This size must then be big enough and λ small enough to support very small share. The value chosen are 1024 for the size of the stride queue, and 1 for λ so that the possible minimal share is equal to $1/1024 = 0.001$, thus to a 0.001% of cpu allocation.

However, this calculation method introduces not negligible errors on process strides, as shown in figure 4.

In this example, both theoretical stride and corrected stride use leads to the expected 80% to 20% cpu allocation to process 1 and 2 respectively, whereas the non-corrected stride leads to a 83% to 17% allocation, with a 3% error. This error may be even bigger depending on ticket allocation (a ratio of 3:2 in ticket allocation result in a 10% error).

3.1.4 Stride Correction

As the ratio λ to the process share is rounded to the nearest integer, the stride error is calculated as follow.

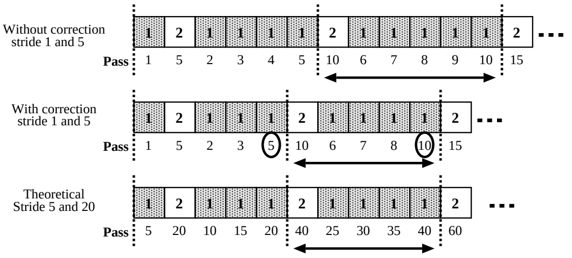

Fig.4 *Stride error effect : two processes with 400 and 100 tickets respectively, corresponding to 80% and 20% of cpu allocation. The scheduling sequence is shown for stride calculation without correction, with our error correction, and with the theoretical stride. The circled pass are incremented with the corrected stride, the repeated sequence are underlined.*

$$Stride\ error = process\ stride - \frac{\lambda}{process\ share}$$

If the stride error is positive, the rounded stride is bigger than the theoretical stride, and thus, over a certain time interval, the process will not receive enough cpu time. On the contrary, if it is negative, the process will receive too much. Few seconds of run can be long enough to start seeing those errors appearing in cpu time allocation to processes.

Let's consider the case where the stride error is positive. Each time the process is scheduled the cumulated error Ce_i is calculated as :

$$Ce_i = stride\ error * n_i + Le_{i-1}$$
$$Le_{i-1} = previous\ stride\ error * n_{i-1} - 1$$

where n_i is the number of times the process was scheduled since the $i-1$ correction, and Le_{i-1} is the part of the error not corrected the previous time. When Ce_i becomes greater or equal to one, the process stride is decremented by one and Ce_{i+1} and Le_i are calculated. For a negative stride error, the stride is incremented by one when Ce_i is lower or equal to minus one, with $Le_{i-1} = previous\ stride\ error * n_{i-1} + 1$.

In the example of figure 4, process 1 with 400 tickets has a stride of 1 with 0.25 error. Its stride is thus adjusted to 2 every time is has been scheduled 4 times since the last correction.

The use of this correction mechanism can change the scheduling order of processes compared with the use of the theoretical stride, leading to a possible increase in the length of the interval of time necessary to obtain fairness. But it has the advantage of greatly reducing the stride calculation overhead, and also makes it bounded : this calculation do not depend on the number of processes in the system.

3.2 Combination of Priority and Stride Scheduling

The basic idea is to distinguish those processes which are compute bound to those which are considered as interactive, and to manage them either with stride and priority scheduling respectively.

3.2.1 Data Structure

As we wish to schedule interactive processes depending on their priority in order to ensure a good response time, a classical structure (array of list) is used. To handle compute bound jobs with stride scheduling, the stride queue structure presented previously is affected a priority, and managed as any other process within the priority queue.

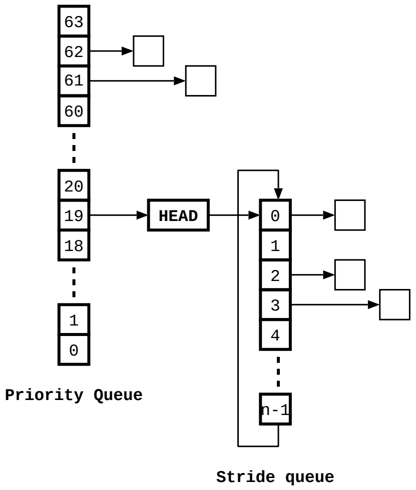

Fig.5 *Ready queue organization : the stride queue is managed as any other process by affecting it a priority.*

The scheduler always compares the stride queue priority with the highest process priority to determine where to get the next process to run, either in the stride queue or in one of the priority queue.

3.2.2 Process Distinction

Classically, processes waiting for events occurrence are given a high priority so that they can be scheduled quickly after waking up. On the contrary, processes which consume up their time slice several times without voluntary relinquishing the processor are given a low priority. Process priorities are used to separate compute bounds jobs form interactive processes. In a first step, process priorities are calculated very simply, using a table shown in figure 6.

High priority processes are given a small time quantum, and low priority processes a longer one

```
const int pritbl[] = {

/* quantum priexp prilslpret prisslpret */  // PRI

    100,      0,        50,          10,         // 0
    100,      0,        50,          11,         // 1
    100,      0,        50,          12,         // 2
...
    100,      8,        58,          28,         // 18
    100,      9,        59,          29,         // 19

/* TS <=> IA LIMIT */

     80,     10,        60,          30,         // 20
     80,     11,        60,          31,         // 21
     80,     12,        60,          32,         // 22
...
     40,     48,        63,          60,         // 58
     40,     49,        63,          60,         // 59
     20,     50,        63,          60,         // 60
     20,     51,        63,          60,         // 61
     20,     52,        63,          60,         // 62
     20,     53,        63,          60,         // 63

};
```

Fig.6 *Example of a table used to calculate process priority and time quantum : the highest priority level is 63, and the lowest 0.*

(a hundred millisecond). Each time a process consumes up its time slice, its priority is decreased (`priexp` priority on expiration), and its time slice increased (`quantum`). A new process is always affected the highest priority (63), and the priority on wake up (`prilslpret` priority after long sleep and `prisslpret` priority after short sleep) depends on the sleep time as well as the priority on the previous run. The IA/TS limit (Interactive/Time Sharing) is the priority level below which a process is considered to be compute bound, a process reaching this priority level is scheduled using the stride queue.

3.2.3 CPU Starvation Avoidance

As all low priority processes are managed with the stride queue, changing the priority of this structure is equivalent to change the priority of all processes in it. To avoid cpu starvation, each time the scheduler chooses a process from the priority queue, the stride queue priority is increased. When this priority is higher than any process in the priority queue, the next process is chosen from the stride queue. If this process uses up its time slice without being preempted by a higher priority process (which will be scheduled with the priority queue), the stride queue priority is decreased to the IA/TS limit priority. On the contrary, if the process is preempted, it is put back at the head of the stride queue with a time slice adjusted to the remaining time of the previous time slice. The stride queue priority is not decreased in this case.

Unlike Unix systems which generally recompute all process priorities regularly to avoid cpu starvation, only the stride queue priority is adjusted depending on its use, and this operation do not depend on the number of processes.

4. Evaluation Results

As mentioned in the introduction, this scheduling method is part of Colonia, a distributed operating system which will run over a cluster of workstations communicating with a special network hardware. As this hardware and also device drivers are not yet implemented, the core part of the kernel of Colonia is actually running over an emulator above Solaris 2.5. The results presented in this section were measured using this emulator.

The next subsection presents some test results of our implementation of stride scheduling. The effect of interactive load on fairness realization will be discuss in subsection 2.

4.1 Compute bound load

In this first example, two users with the same number of tickets are both running compute bound jobs. User A runs two processes with 100 and 400 tickets respectively, user B runs only one process which sleeps on a regular basis waiting for events to occur. Figure 7 shows the load of each process in percentage of the cpu time used over one second intervals (considering the ticket allocation in this case and the time slice of a hundred millisecond for processes scheduled using the stride queue, fairness should be ensured over one second intervals).

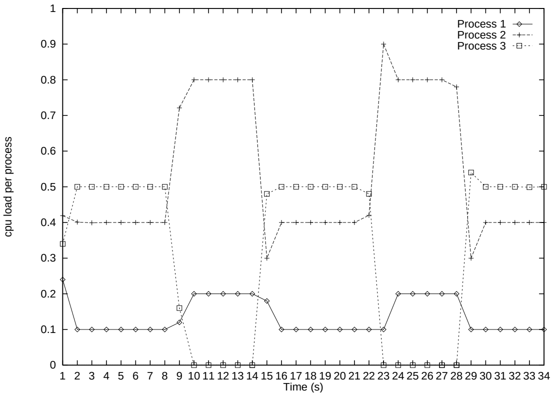

Fig.7 *Two users with the same number of tickets are running two and one compute bound jobs respectively. Processes 1 and 2 of user A have 100 and 400 tickets respectively, process 3 of user B receives half of the cpu time when it is runnable, process 1 and 2 sharing the other half with a 1:4 ratio. This graph represents the cpu load for each process over 1 second intervals.*

When processes start execution, they get a high priority and thus are scheduled using their priority. This can be seen at the beginning of each load graph, fairness is not achieved. But as these processes use up their time slice several times without sleeping, they are considered as compute bound by the scheduler which then manages them using the stride queue.

When process 1 wakes up (15 and 29 seconds), it gets a high priority, preempting process 2 and 3 which then can't get their share of cpu. Fairness is again ensured within one second interval, as process 1 enters again the stride queue. When process 3 sleeps, process 1 and 2 shares are recomputed to redistribute computing time. This adjustment introduces some errors in fairness for a short interval, as shown in Figure 8.

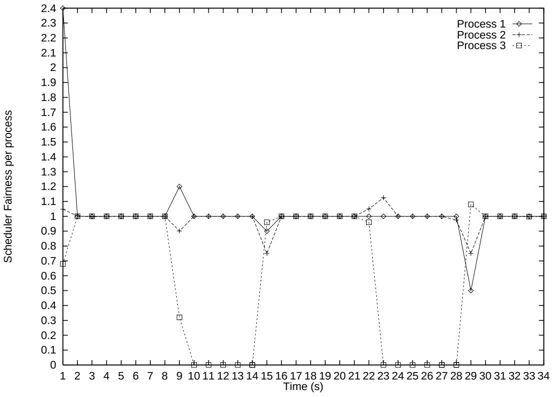

Fig.8 *Scheduler fairness for the first example over 1 second intervals.*

The scheduler fairness was measured using the following formula,

$$Fairness = \frac{Ct}{(Ct + Wt) * process\ share} \quad (3)$$

where Ct is the cpu time received and Wt is the time waited in runnable state in either the stride queue or the priority queue.

In both figures 7 and 8, we can see that when process 1 goes to sleep (9 and 23 seconds), the stride adjustment introduces some errors in cpu allocation distribution, but within one second interval, fairness in ensured again.

As expected, our implementation of stride scheduling can ensure fairness very quickly. In this example, with a hundred millisecond time slice, fairness is guaranteed within one second intervals. Our stride correction mechanism do not affect the length of the interval of time necessary to ensure fairness.

4.2 Interactive and Compute bound load

In the second example of figure 9, user A is running the same compute bound jobs as in the first example on a node where user B is now running interactive processes. These processes always sleep before they enter the stride queue, so that they are always scheduled on a priority basis.

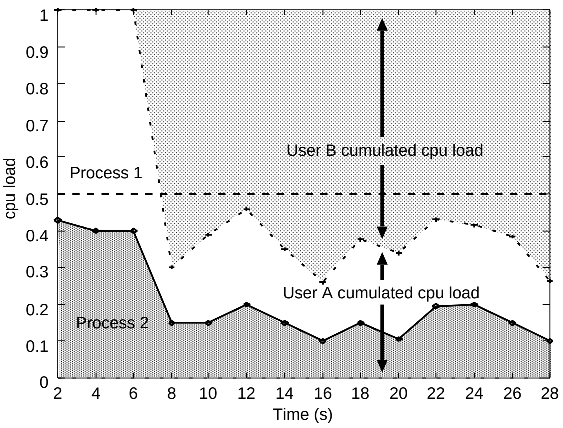

Fig.9 *Cpu load for user A processes over 2 second intervals under a high interactive load run by user B (not represented).*

In this case, the load of both compute bound processes decreases as soon as interactive processes start execution (after 6 seconds). As the scheduler uses more often the priority queues to choose processes, processes 1 and 2 do not receive enough cpu time, fairness is not ensured. However, the relative load between these two processes is preserved, as the cpu starvation algorithm and the preemption control prevents any process in the stride queue from getting more cpu time than its share (relatively to other processes in the stride queue), and ensuring also that the stride queue is used regularly to schedule processes.

This examples shows that fairness is not achieved when a heavy interactive load is concurrently running on the same node. Because priorities are calculated statically without using the share of cpu time allocated to processes, as in classical priority scheduling, a user with many interactive processes can still get more cpu time.

5. Conclusion and Future Work

We have presented our implementation of stride scheduling which allow an fair allocation of cpu time using tickets allocated to users and processes, with a very low overhead. Preliminary results have shown that fairness is ensured for compute bound processes over very short time intervals, even in the case of dynamic change in cpu share allocation. The combination of this method with a more classical priority scheduling scheme ensures a good response time to event occurrence for interactive processes. In this case, preemption of compute bound jobs by high priority interactive processes is necessary, resulting in an acceptable degradation of fairness. However, as priorities are affected depending only on the run time behavior of processes, this degradation can persist so that processes can get more than there fair share, whereas it should be limited to few seconds.

The next steps of this work will investigate method to separate more efficiently processes depending on their behavior, and to penalize of processes (and users) receiving more than their fair share by either introducing a priority correction depending on the share and on the previous cpu usage, or by using more intensively the stride scheduling method.

More work also needs to be done on the definition of interactiveness. For example, a compute bound job waiting on a semaphore could be seen as an interactive process. We need to raise parameters allowing the scheduler to differentiate I/O bound processes (typically user interfaces, text editors, and more generally processes using external devices I/O) to compute bound processes and synchronized parallel applications.

A more efficient resource usage could be also implemented using a system wide ticket inheritance mechanism. Indeed, tickets of sleeping processes are unused and thus wasted, whereas for example applications could exchange tickets to reduce the wait time on synchronization, and processes holding critical resources could be scheduled more often to relinquish these resources faster.

Extensions of stride scheduling to multimedia real-time scheduling will also be investigated, because a constant service rate can as well be implemented using this method.

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)#10558045, and (C)#09680334 from the Ministry of Education, Science, Sports and Culture.

References

1) Jean Bacon: Concurrent Systems - Operating Systems, Database and Distributed Systems : an integrated approach, *Addison Westley*, Second edition (1997).

2) A. Goscinski: Distributed operating systems - the logical design (1991).

3) Maurice J. Bach: The design of the Unix operating system (1986).

4) Amnon Barak, Shai Guday, Richard G. Wheeler "The Mosix distributed operating system : Load balancing for Unix",*Lecture Notes in Computer Science*, vol. 672, (Springer-Verlag, Berlin, 1993).

5) A. Silberschatz, J. Peterson, P. Galvin "Operating System Concepts", Third edition (1991).

6) Prabhat K. Andleigh: Unix System Architecture (1990).

7) AT&T Unix System V Release 4 *Programmers Guide : System services and Application Packaging tools* (1990).

8) S. Khanna, M. Sebree, J. Zolnowsky: Real-time Scheduling in SunOS 5.0, *USENIX Conference Proceedings* (winter 1992).

9) Carl A. Waldspurger, William E. Weihl: Stride Scheduling - Deterministic Proportional-Share Resource Management, *Technical memorendum MIT/LCS/TM-528, MIT Laboratory for Computer Science* (June 1995).

10) Carl A. Waldspurger, William E. Weihl: Lottery Scheduling - Flexible Proportional-Share Resource Management, *Proceedings of the first Symposium on Operating Systems Design and Implementation* (1994).

11) Andrea C. Arpaci-Dusseau, David E. Culler: Extending Proportional-Share Scheduling to a Network of Workstations, *Computer Science Division, University of California, Berkeley* (Undated).

12) Raymond B. Essick: An Event-based Fair Share Scheduler, *Usenix* (Winter 1990).