The Local Scheduler of the Distributed Operating System Colonia
分散OS Coloniaのローカル・スケジューラ

Damien Le Moal

2000, February 9

内容梗概

ワークステーション・クラスタは並列処理を行う分散システムとして発達してきており、各計算機の負荷は高くなってきている。しかしながら多くのユーザが様々なアプリケーションを実行するにもかかわらず、システムは昔と変わらずたった一人のユーザのことしか考えられていない。したがって複数のユーザに対して公平なCPU時間を与えるような新しいスケジューリングが必要である。クラスタ全体で公平なCPU時間を与えるため、ローカルとグローバルという二つのスケジューリングポリシーを使用する。各ノードのローカル・スケジューリングはそのノードのローカルプロセスを公平に実行し、クラスタ全体の公平なスケジューリングは動的なプロセスマイグレーションで実装される。

現在プライオリティ・スケジューリングの様な良く使われているスケジューリング方法は様々な種類のプロセスを効率的に実行出来るが、スケジューリングの公平性は保たれていない。一方、Fair-Shareスケジューリング手法の場合ではプロセスの実行順序の制御がうまく出来ないためI/Oに依存するようなプロセスを効率的に実行できない。

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。プライオリティ・スケジューリング法を使用して正確なプロセスの実行順序を出来る。各プロセスが割り当たれたCPU時間によてStride Schedulingを利用して公平なスケジューリングを実現できる。

この拡張されたFair-Shareスケジューリング方法を評価した結果、普通の公平なスケジューリングよりもCPU時間の割り当てが正確になり、プロセスの実行順序を制御することができ、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できることが分かった。スケジューリングのオーバヘッドは限界に達し、プロセスの数によらず一定となった。また、負荷の高い様々な種類のアプリケーションの場合でも効率よく正確に公平なスケジューリングを実現出来る。

Abstract

As distributed systems, and in particular workstations clusters emerge as a viable platform for parallel processing, the workload of desktop computers is changing. Whereas classical systems generally have to handle mainly interactive sequential processes of a single user, in such systems processing elements must support a variety of applications by a great number of users. A new scheduling method is needed to support such workload and especially to implement a fair-share allocation of processing time among users. Fairness should be first implemented locally to each processing element of the cluster (local scheduling), global scheduling ensuring global fairness through process migration.

Classical scheduling methods like priority scheduling can efficiently support the execution of various type of applications, but do not allow to implement fair-share scheduling simply. On the contrary, efficient strict fair-share scheduling scheme exists but do not provide good results for I/O bound and interactive processes execution, because not considering basically the run-time behavior of processes.

In this report, it will be shown that extending the fair-share scheduling method called stride scheduling to execute more efficiently interactive and I/O bound workload can still fairly allocates processing time among users. A novel implementation of stride scheduling is presented. This implementation is enhanced using a more classical priority scheduler to support both compute bound and interactive applications. A precise control mechanism is implemented to make this method more flexible and adaptable to a wide range of workload.

Evaluation results show that the fair-share allocation of CPU time among users and processes is improved compared to strict fair-share scheduling. It is also shown that the scheduling overhead is bounded and does not depend on the number of runnable processes. Finally, some experiments show that this scheduling method can precisely handle even a high workload of a mix of applications in a multi user environment.

Contents

1 Introduction

Personal computers and workstations connected through a local area network is a common environment to share physically distributed resources to reduce the cost of providing each user with a wide range of services. The increasing performances of processors allows to run effectively a wide range of applications. Also, as mainly all machines in such environment deal with only the processes of a single user, users are more able to predict the response time of their applications. Such classical network of loosely-coupled workstations can match the needs of most users. However, the non-transparent access to physically distributed resources makes difficult to share data between parallel applications components distributed on several computers, allowing only a coarse-grain parallelism in program execution. Moreover, it is difficult to evenly balance the workload of all machines on the network. Some machines may be idle while other overloaded.

Distributed operating systems address these problems by more tightly coupling computers in order to implement an homogeneous image of a single computer system, and to extend the services provided by each machine of the network. 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 constituting the system from mainly sequential processes of a single user to a variety of applications executed by a large number of users.

Computer Colony is a cluster of workstations, actually developed in our laboratory, which uses a dedicated network device and a specialized distributed operating system. The high-speed and low-latency network interface supports hardware cache coherent distributed shared memory and allows user level communications, reducing the overhead of sharing physically distributed data of parallel application components within the cluster. The specialized distributed operating system called Colonia implements from the user point of view an image of a massively parallel processor supporting fine-grain parallelism.

In such environment, it is desirable to guarantee that each user can be allocated fairly resources and in particular processor time. Most scheduling policies, like those used in UNIX like operating systems, permit a user running many processes to receive more processor time at the expense of users running only few. 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. The system should allocate CPU time among users and processes so that one user cannot degrade another user process execution by running more processes. In particular, interactive and I/O bound application should be guaranteed a good response time. 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.

To be able to schedule fairly applications throughout such system, one should first be able to implement fair-share scheduling at the level of each processing element. Recent research works have produced several specialized scheduling methods allowing a fair-share allocation of processing time. In a multi-user environment, all users can receive the same share of CPU time regardless of the number of processes they execute. But because such methods do not consider the run-time behavior of processes when choosing a process for execution, they only performs well for compute bound applications. The response time of interactive and I/O bound processes is degraded as such processes are not executed quickly on wake-up. Even though some improvements are possible, this usually increases the overhead of those methods and makes their implementation more difficult. On the contrary, general scheduling policies do support with simple algorithms interactive and I/O bound workload while providing a good throughput to compute bound applications, with a very small overhead, but do not allow to implement a fair-share allocation of CPU time simply.

The approach taken to solve this problem in Colonia is to combine a fair-share scheduling method with a more classical scheduling policy providing the desired control over process scheduling order to execute efficiently interactive and I/O bound applications. It will be shown that this approach is viable and provides within a small overhead a precise control over CPU time allocation without degrading the response time of user interactive processes. Moreover, the scheduling algorithm is kept simple to allow a great flexibility.

The rest of this report is organized as follows. Chapter 2 discusses the relative performance of various scheduling policies and present some fair-share scheduling methods. Colonia scheduling policy and its implementation are presented in chapter 3, followed by evaluation results in chapter 4. Finally chapter 5 concludes this report.

2 Background and Related Work

One should be careful to not confuse scheduling policy and scheduling mechanism. The scheduling policy defines how the CPU scheduler of an operating system should choose processes which are ready to run for execution. The scheduling mechanism implements the policy defined. If these two abstractions are carefully separated, it may be possible to implement several policies with the same mechanism. This section presents various scheduling policies and discusses their relative problems. Scheduling mechanism implementing the policies presented are considered only if necessary.

First some metrics generally accepted to measure performances of CPU schedulers are explained, followed by a discussion on process execution representation. Commonly used scheduling policies are then presented, fair-share scheduling methods are discussed next. Finally related works are presented and discussed. Real-time scheduling and multi-processor scheduling will not be considered here.

2.1 Scheduling Algorithms Evaluation

Many criteria have been proposed to evaluate and compare relative performances of scheduling algorithms. Results may vary depending on the criteria chosen, but mainly the following properties are usually observed.

Throughput The throughput is the measure of the average number of processes completing execution per unit of time.

Turnaround time It is the total execution time of a particular process from the submission time to the completion time. It is the sum of the CPU time, sleep time and wait time in runnable state.

Wait time The scheduler does not affect the CPU time needed by a process to complete its execution, however, it does affect the time a process waits for the CPU in runnable state.

Response time It is important for interactive systems to ensure a good response time which is the time needed to start responding to an event.

And of course, the CPU utilization, as the CPU should be kept busy as much as possible. It is desirable to minimize turnaround time, wait time and response time, whereas the throughput and CPU utilization should be maximized. However, more than extreme values, the variance of these metrics is important, in particular for interactive systems. Indeed a predictable system is one which can provide some guarantees that one of these criteria will not exceed a maximal or minimal value close to the average. Such system can often provide a better quality of service (QoS) than one faster in the average but with important variations.

2.2 Process Execution Representation

In this section, abstractions used throughout this report are presented, in particular the process execution model used.

A process execution can be represented as cycles of processor use (CPU burst) followed by I/O completion wait time (I/O burst). A process always starts with a CPU burst, followed eventually by an I/O burst, the last CPU burst may allow a process to complete its execution. The success of choosing a scheduling policy depends greatly on the observed distribution of these CPU burst and I/O burst cycles on the target system.

As shown in figure 1, a process can be basically assigned one of the three states Run, Ready and Sleep. The Run state is assigned to a process running on a processor. The ready state is assigned to processes not logically blocked but not running, waiting to be allocated a processor in the ready queue. Finally, the sleep state is assigned to processes which performed blocking system calls, waiting for the event related to this call to occur. A process having a CPU burst is thus either in the run or ready state, whereas I/O bursts always correspond to the sleep state.

Transitions between these states correspond to one of the three basic functions defined by schedulers. The `schedule()` functions preempts the currently running process and chooses the next process to run between the runnable ones. The `sleep(event)` function assigns the sleep state to the calling process, waiting for `event` to occur. The `wake_up(event)` functions put into the ready queue and assigns the ready states to processes which were waiting for `event` to occur. The scheduling policy defining how to choose processes for execution is thus mainly implemented within the `schedule()` function.

The next sections present some of the commonly used scheduling policies and discuss fair-share scheduling.

2.3 Common Scheduling Policies

Operating systems, in particular several versions of the Unix operating system, tries to execute efficiently various type of applications 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 allowing to 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 these policies, fair-share scheduling will be presented next.

2.3.1 Short-Job-First Scheduling

The short-job-first policy (SJF) was first designed for batch systems where processes (jobs) were running until completion before a new process was chosen. However, this policy can be extended to multiprogrammed time-shared systems by considering that a “job” is in fact a CPU burst of a process. The SJF policy always chooses the process with the shortest next CPU burst. Figure 2 shows an example where four processes with different CPU bursts length are scheduled using a random order, let say the first-come-first-served (FCFS) policy, and with the SJF policy.

The resulting average wait time under the FCFS policy is much higher than the one provided by the SJF policy. In fact, the SJF policy is provably optimal for interactive systems as interactive and I/O bound processes which generally have short CPU bursts will be scheduled quickly improving their response time. However, it is difficult to implement because the CPU burst of a process is not generally known in advance and must thus at least be predicted.

2.3.2 Round-Robin Scheduling

A much more 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. Unlike the SJF policy, RR scheduling is preemptive.

This policy performance depends greatly on the length of the time quantum. If it is too long, RR degenerates to FCFS and provides poor performance results for I/O bound processes, whereas compute bound processes turnaround time 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 SJF policy. However, if the time quantum is too short, this policy is then called *processor sharing* and is seen from the user point of view 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. Processor sharing is never used because the ratio of time spent performing context switches becomes very high, increasing system overhead.

2.3.3 Priority Scheduling

Whereas the RR policy makes the assumption that all processes are equal, priority scheduling associates a priority (i.e. 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 queues of processes of equal priority are scheduled using different policies, priority scheduling being used between queues. Figure 3 shows an example of such structure.

Several methods can be used to calculate priorities, depending on various parameters. A widely used

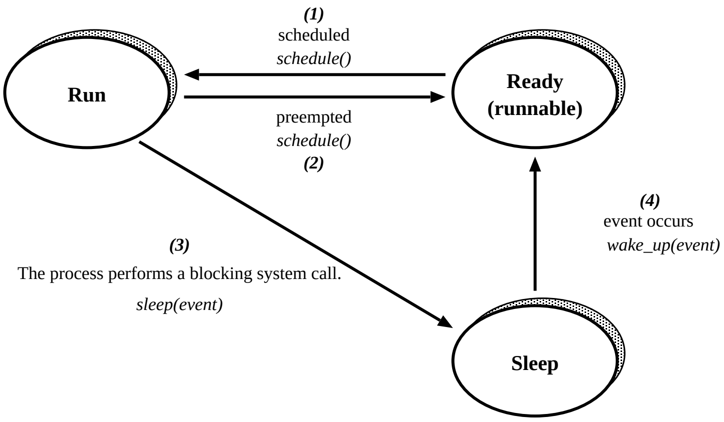

Figure 1: *State transition graph.*

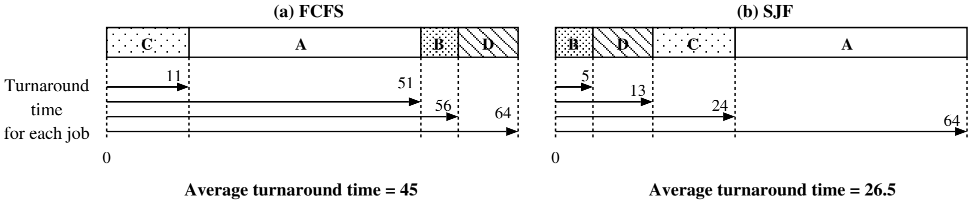

Figure 2: *The Short-Job-First policy executes processes with short CPU burst first.*

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 releasing voluntary the processor often 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 ([3] and [2]) 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.

2.4 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.

Fair-share (or proportional fair-share) scheduling 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.

2.4.1 Priority-Based Fair-Share Scheduling

The first fair-share schedulers were priority based. In this scheme, priorities are calculated depending on the share of CPU time allocated to users or processes, or to groups of user or processes, and on the past CPU usage. Among famous work in this area are the fair-share scheduler of AT&T [4], the work presented in [6] or the event based fair-share scheduler described in [7].

Priority-based fair-share schedulers can provide good results, but usually realize a fair allocation of CPU time over long time intervals. Moreover, priority calculation methods used are generally rather difficult, making such scheme difficult to tune and its implementation

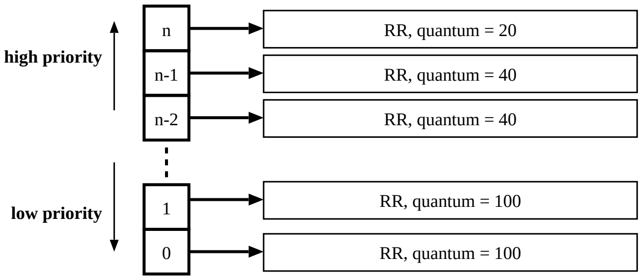

Figure 3: *Multilevel feedback queue scheduling using priority scheduling.*

not flexible. The work presented in [6] recomputes all process priorities every 4 seconds, with fairness realized over hours or days. In the work presented in [7], priorities are recalculated on event occurrence and as generally one event (I/O completion, preemption, ...) only concerns one or few processes, 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 considering a lot of parameters to penalize or boost processes execution depending on their CPU usage.

2.4.2 Lottery Scheduling

More recent works in the area of fair-share scheduling have produced several interesting methods, among them is lottery scheduling [8]. The basic algorithm is rather simple. The scheduler chooses a ticket randomly using a lottery, the process owning the winning ticket is executed for one time quantum.

Figure 4 shows an example where three processes (1 to 3) compete for the processor with a ticket allocation ratio of 4:3:1. Process 1 is thus allocated $4/(4+3+1) = 50\%$ of the CPU time, process 2 and 3 sharing the other half with respectively a share of 37.5% and 12.5%.

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 long.

2.4.3 Stride Scheduling

Stride scheduling [10] 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 the stride of a this process each time it is scheduled. The scheduler always chooses the process with the smallest pass for execution.

Figure 5 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. As process C has the smallest pass (6), it is next scheduled, followed by process B and again process C. Over the represented sequence of this example, process C is scheduled five times, process B three times and process A only two times, resulting in the desired 2:3:5 ratio. At the end of this sequence, all processes pass are equal, the same pattern will then be repeated again. Stride scheduling thus allow to achieve fairness over very short time intervals.

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.

$$process\ stride = rint(\frac{2^{20}}{process\ share})$$

Where `rint()` is the function rounding a floating point to the nearest integer. The constant 2^{20} 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 undetectable practically.

The ready queue structure used is a sorted list in increasing pass order. The resulting overhead of this method is thus not negligible for systems with a high load. It is in $O(n)$ where n is the number of processes, at best in $O(log(n))$, depending on the sort algorithm used. Moreover, as stride scheduling, like lottery scheduling,

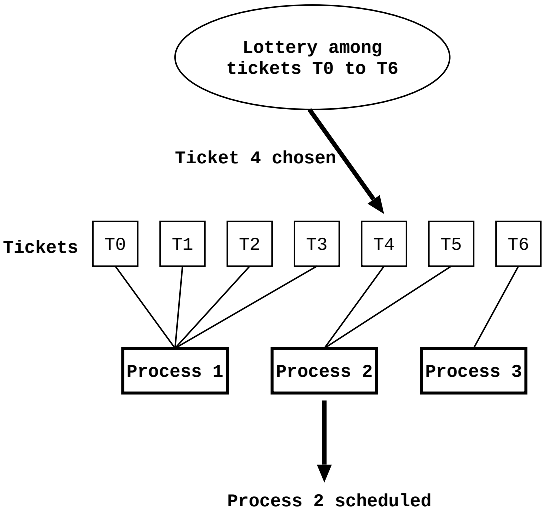

Figure 4: *Lottery scheduling principle.*

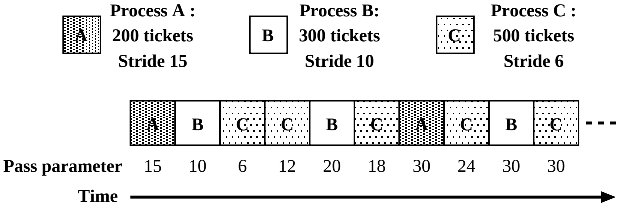

Figure 5: *Stride scheduling.*

does not consider the run-time behavior of processes, it suffers from several problems.

- 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, 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 next section.

2.5 Related Work

Recent work (presented in [11] and [9]) 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 [11] 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 : in this case, the system gives more tickets to a sleeping process in proportion to the slept time. Another improvement presented in [10] 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 increased less so that they are scheduled sooner.

However, all these methods do not implement a preemptive scheme, increasing the dispatch latency of processes on wake up. Also, a heavy interactive 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 adjust-

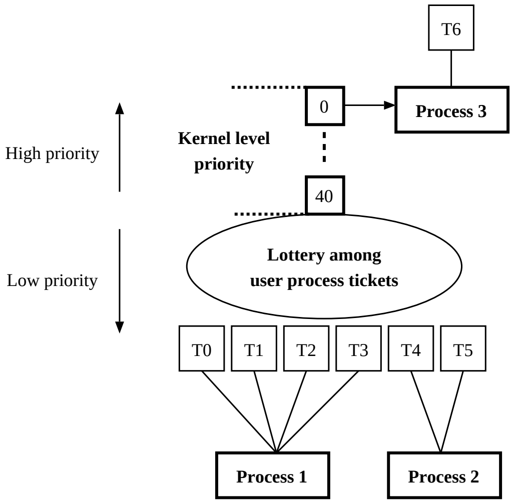

Figure 6: *Hybrid lottery scheduling combines the FreeBSD system level priority scheduling with an improved lottery scheduler.*

ments. 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 [9] uses this modification but goes further and implements a preemptive scheduling. It combines the system level priorities of the FreeBSD scheduler [5] 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.

The next section will presents our solution to improve fair-share allocation of CPU time and responsiveness of interactive processes using a simple implementation of stride scheduling.

3 Enhanced Stride Scheduling

To be able to schedule fairly applications throughout the distributed system Computer Colony, the distributed scheduling policy is separated into local and global scheduling. Local scheduling should implement a fair-share scheduling method at the level of each processing element, considering only the local workload of each node. The global scheduling policy defines how local workload should be distributed in the system to improve process execution and resource use, but also to guarantee a global fairness.

This chapter presents in detail the scheduling policy used and its implementation. First, our approach of fair-share scheduling is discussed. The share calculation method used is presented next, followed by the presentation of the implementation of stride scheduling. The scheduling policy and the general scheduling algorithm are then presented.

3.1 Overview

These solutions are thus mainly based on difficult mechanisms, so that with an invariant scheduling mechanism (implementing a single policy) all processes can be allocated their fair-share of CPU time.

Indeed, strict fair-share scheduling methods fail at efficiently executing I/O intensive and interactive processes because they do not consider their run time behavior when choosing processes for execution. Improving the responsiveness of such scheduling methods by a dynamic adjustment of ticket allocation to process increases overhead and is not efficient for a high load of interactive applications. Also, processes which where granted a temporary ticket boost for having slept may be scheduled more, but are not guaranteed at all to be scheduled quickly, that is to preempt the running process and to run soon after waking up.

The approach taken to solve this problem is here different. The share of a process is not changed depending on its CPU usage but the scheduler behaves differently. The core idea is to guarantee both a good response time and fairness is to schedule I/O bound and interactive processes ahead of compute bound processes using priority scheduling, until either those processes sleeps or are allocated their fair-share of processor time. If those processes are still runnable after receiving their share of

CPU time, they are scheduled with stride scheduling.

A mechanism is thus needed to precisely control the relative importance of priority and stride scheduling so that fairness can still be guaranteed while improving response time with priority scheduling. This mechanism will be presented with the scheduling policy. The next section first introduces the share calculation method used.

3.2 Ticket Allocation and Share Calculation

The scheduler should first allocate CPU time fairly among users. However, a hierarchical definition of shares is desired to not only define the share of a user but also to allow users to allocate different CPU time share to their processes.

This section first presents the ticket allocation scheme used. The calculation method defining process share is then explained.

3.2.1 Tickets

Scheduling of process in Colonia should be first fair from the user point of view, that is users should be allocated some tickets at login time to determine the share of CPU they can use. However, a user may also desire to weight differently his processes. Processes should thus also be allocated tickets to determine the ratio of the share of their user they can use. Such scheme is usually called hierarchical fair-share. Two different sorts of tickets are used : user and process tickets.

Process tickets Process tickets are allocated to a process by its user, and are used only to determine the share of the process within the share allocated to its owner. Process tickets are said to be active if this process is runnable or running in the node considered.

User 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 runnable or running process in the node, i.e., if this user has a total of active process tickets greater than zero.

To reduce the overhead, the total number of tickets and active tickets of each type can be maintained simply on event occurrence, for only users and processes concerned by this events. Typically, event needing tickets adjustment are events implying a change on the state of user processes, for example when a user logs in or out the system, or more generally when processes enters and leaves the ready queue. Also, setting no limit to the total number of tickets in the system simplifies greatly the management. Tickets should thus be used by considering only their relative number. This is explained in the next section.

3.2.2 Share Calculation

As the scheduler only considers processes, it is only interested by process share. As no maximum is set to the total number of tickets of each type in the system, shares are calculated using tickets ratio. The minimal share of CPU time a process can be allocated is calculated using the tickets of this process and of the user owning it. The minimal share of the process k of user u, is calculated with the following formula.

$$\begin{aligned} & \mathit{process\ minimal\ share} \\ = \ & \mathit{user\ u\ minimal\ share} * \mathit{process\ k\ ticket\ ratio} \\ = \ & \frac{tu_u}{\sum_i tu_i} * \frac{tp_{u\,k}}{\sum_j tp_{u\,j}} \end{aligned}$$

Where $\sum_i tu_i$ and $\sum_j tp_{u\,j}$ are respectively the sum of user tickets and the sum of process tickets for user u. However, all processes are rarely runnable at the same time. To adjust the share of processor time a process can be allocated depending on the current workload of the node, it is better to calculate the share using active tickets. This share is called the active share and calculated with the same formula as the minimal share considering only active tickets sums.

$$\begin{aligned} & \mathit{process\ active\ share} \\ = \ & \mathit{user\ u\ active\ share} * \mathit{process\ k\ active\ ticket\ ratio} \\ = \ & \frac{tu_u}{\sum_i tau_i} * \frac{tp_{u\,k}}{\sum_j tap_{u\,j}} \end{aligned}$$

Where $\sum_i tau_i$ and $\sum_j tap_{u\,j}$ are respectively the sum of active user tickets and the sum of active process tickets for user u.

As process active shares are calculated using active tickets sums, they are greater or equal to the minimal share. Because active tickets are maintained on event occurrence, the minimal share and the active share are recalculated both when a process is scheduled, to take into account any possible modification of ticket allocation. As a result, a single user running some processes alone in a node can receive all the CPU time whatever its ticket allocation is. As soon as another user process wakes up or is started, active tickets are updated so that active shares can be recalculated. CPU time is always shared among all runnable processes depending on their relative ratio of user and process tickets.

Figure 7 shows an example where two users A and B with the same number of tickets are both executing 2 processes. Process 2 of user A is sleeping waiting for I/O completion, thus its tickets are not active. Process

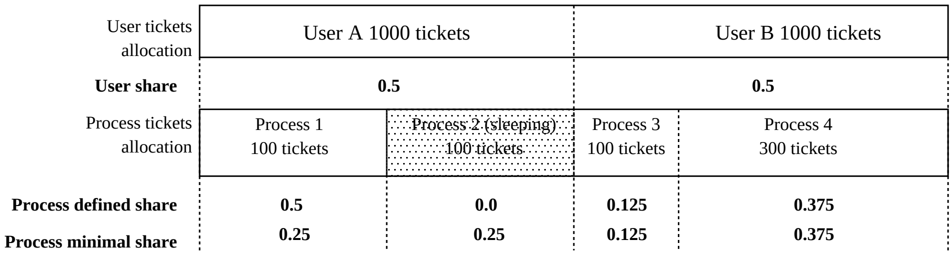

Figure 7: *An example of ticket allocation with the resulting CPU share for each user and process.*

1 can then uses all the active share of user A. On the contrary, as all processes of user B are runnable, process 3 and 4 are assigned a ratio of user B active share, the ratio of user B active tickets they own.

The next section presents our implementation of stride scheduling which directly uses the active share to compute the stride of each process.

3.3 Stride Scheduling Implementation

Stride scheduling is an efficient fair-share scheduling method allowing to realize fairness simply and over short time intervals. It is however desirable to reduce its overhead and to allow process preemption while preserving its characteristics.

Our implementation of stride scheduling uses a data structure avoiding to sort processes into increasing pass order and is also modified to support preemption without degrading the strict fair-share allocation of CPU time provided by the basic algorithm. This section first presents the data structure used, stride calculation is discussed next.

3.3.1 Stride Queue Structure

The data structure used to choose processes with stride scheduling 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 get operations are thus done in the head list. Figure 8 shows the example of figure 5 using this data structure. The stride queue state is represented at different instant of the represented scheduling sequence.

The head pointer is advanced 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. Sorting a queue of processes is avoided as lists are naturally ordered in increasing pass order from the list pointed to by the head.

A different "put" operation is used for preempted processes. Such processes 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.

However, another stride calculation method is needed. Indeed, the original implementation uses a method which generates big integers, allowing to differentiate processes with a close share using a greater difference in their stride. Such stride values cannot be practically used to calculate an array index, because the size of this array would be too big. Smaller values of process strides are necessary, but yet supporting a precise definition of shares. The calculation method used is presented in the following two sections.

3.3.2 Stride Calculation

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

$$process\ stride = rint(\frac{\lambda}{process\ active\ share})$$

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 active share fraction numerator in the active share calculation formula. This is a too costly operation to be calculated each time the scheduler recomputes strides. 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

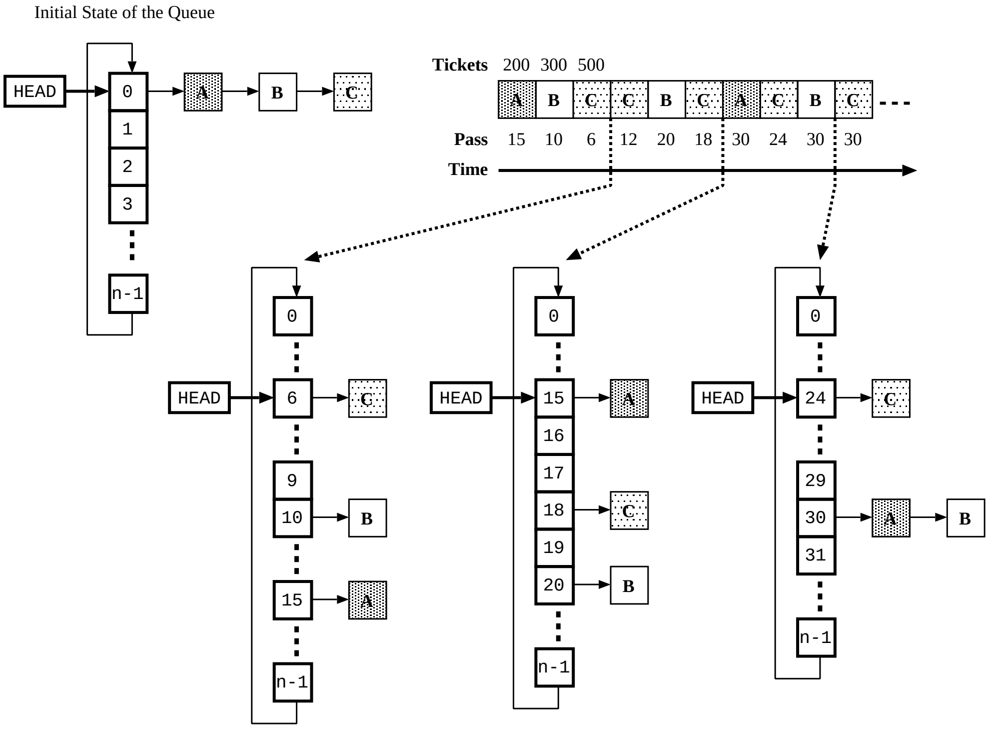

Figure 8: *Example of figure 5 using our implementation of stride scheduling.*

to the size of the stride queue. This size must then be big enough and λ small enough to support very small share. The chosen values 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% share of CPU time allocation.

Because active tickets sums and active shares are updated both on event occurrence, the stride of a process needs to be recalculated each time it is scheduled to take into account any possible change in ticket allocation. This calculation method generates stride values usable as a displacement index into the stride queue, but the round operation introduces not negligible errors. Stride value correction is discussed next.

3.3.3 Stride Correction

As shown in figure 9, depending on the ticket allocation to users and processes, the ratio $\lambda/process\ active\ share$ may not be an integer. The round operation thus introduces errors in the stride calculation. As stride values are small integers, this error can represent a not negligible ratio of the stride value.

In this example, both theoretical stride and corrected stride use leads to the expected 80% to 20% cpu allocation for 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 results in a 10% error).

As the ratio λ to a process active share is rounded to the nearest integer, the stride error is calculated simply as follows.

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

If the stride error is positive, the rounded stride is bigger than the real stride (the not-rounded stride value), and thus over a certain time interval, the process will not receive enough cpu time. Indeed, its rounded stride is bigger than its real stride, resulting in bigger displacement into the stride queue. On the contrary, if it is negative, the process will be allocated too much CPU time. A few seconds run can be long enough to start seeing those errors appearing in CPU time allocation.

Let us 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 from the $i-1$ correction, and Le_{i-1} 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 recalculated.

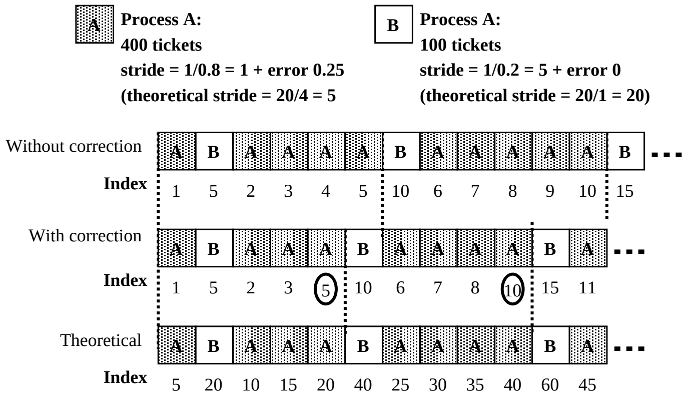

Figure 9: *Stride error effect on the allocation of CPU time using stride scheduling. The circled indexes are incremented with the corrected stride.*

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 9, process 1 with 400 tickets has a stride of 1 with a 0.25 error. The index of the list where process 1 is put is thus normally increased by one each schedule operation, but increased by 2 when it has been scheduled 4 times from the last correction. If a process sleeps waiting for I/O completion, the recorded stride error is erased and the correction started again from the beginning if it enters again the stride queue.

This implementation allows to reduce the overhead of stride scheduling by suppressing the sort of the stride queue. However, it still only implements a strict fair-share allocation of CPU time. The next section discusses in more detail the overall processor time allocation policy used to improve the response time of I/O bound processes while being fair.

3.4 Scheduling policy

To improve both fairness and response time of interactive processes, processes should be scheduled quickly on wake up after waiting for I/O completion and be allocated at least their active share during their CPU burst. As they were sleeping, they should even be allowed to receive temporarily more than their active share to compensate the slept time. A ticket boost does in fact results in this behavior, but scheduling should also be preemptive to minimize the wait time in ready queue on wake up.

Such behavior is desirable to emulate the short-job-first policy because allowing a process to complete a short CPU burst on wake up ahead of other runnable processes. It can also improve kernel contention as kernel shared resources hold by processes sleeping can be released faster.

As the CPU time allocated may be higher than the active share of a process on a certain interval of time, depending on the past CPU usage, basically two cases can be considered.

- The process can complete its CPU burst within the allocated share of CPU time on wake up, sleeping before consuming more than its active share.
- The process has a longer CPU burst and is still runnable after receiving its active share or more. In such case, the boost resulting by the over share allocation should be penalized with a "*unboost*" to preserve the overall fairness, so that processes which used the cpu cycles unused when the process was sleeping are not also 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 in the order of the second. A sweeping window of variable time length 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 10 shows the allocated share of an hypothetical I/O bound process measured over such sweeping window. The two cases discussed above are represented. When the process wakes up the first time, it is allocated more than its active share, which is acceptable considering that it was sleeping for a long time before. On longer CPU bursts, it is however desirable to implement a stricter fairness, so that the process is penalized ("*unboosted*") after receiving its boost on wake up.

Measuring the allocated share over a sweeping window can provide a precise mechanism to control process

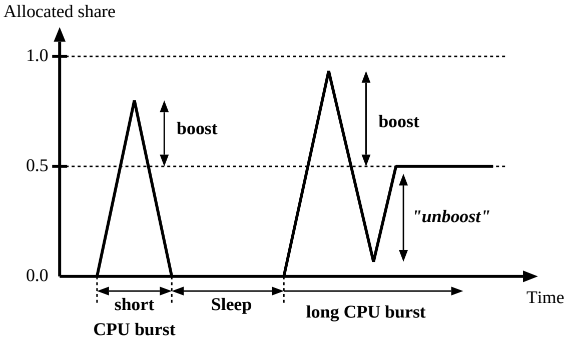

Figure 10: *I/O bound processes must be boosted on wake up. When having a longer CPU burst, the process must be penalized to preserve fairness.*

boosts and penalties. As a result, slept time shorter than the window time length can be easily overlapped by a not penalized boost. The boost can be used also as a compensation allowing processes sleeping for I/O to receive their active share of CPU time.

Figure 11 shows such behavior. The allocated share of the represented process decreases as it sleeps. On wake up it is boosted without penalty. The resulting overall share allocation should match over the sweeping window the active 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 "*unboosted*" so that other processes are not penalized for having used the share of processor time unused by sleeping processes. One should notice coherence between this policy and the active share calculation. As active shares specifies the share of CPU time a process can be allocated during a certain interval, a synergy must exist between boost/penalty and active share calculation.

The time length of the sweeping window can also be used to control the relative strength or weakness of the fair-share allocation. Long window may allow bigger boost, resulting in an unstable distribution of processor time among runnable processes. On the contrary, very small windows will degrade the scheduler behavior to the one of strict fair-share scheduling methods.

The next section presents in more detail the implementation of the accounting window to precisely control CPU time allocation. The scheduling algorithm implementing this policy will be discussed next.

3.5 CPU Usage Accounting

To be reactive to process execution behavior, the scheduler should not only maintains CPU usage information over the life time of processes, but needs also to measure the share of CPU time a process was allocated recently. As explained previously, using a sweeping window can allow a precise accounting of CPU usage to control boosted CPU time allocation on wake up or penalties.

Basically three states are of interest in this case, when a process is running, waiting to be scheduled or sleeping waiting for I/O. On event occurrence, like preemption or execution of a blocking system call, a process switches from the running state to either the waiting or sleeping state. As shown in figure 12, the sweeping window records the time interval of each state and if the process was using the processor or not during this state (a boolean). As on each event occurrence a process active share is also recomputed, the active share during this state is also recorded.

To control more precisely the combination boost/unboost or sleep/boost explained previously, the allocated CPU time over the first half of the window (most recent allocation) is also measured. For example, in figure 13, the boost is measured over the first half of the window. As the process goes on executing, the boost recorded is shifted to the end of the window. The unboost can be calculated precisely over the next first half of the window to allocate the active share over the total window.

On the contrary, if a process sleeps, the allocated share will decrease on the first half of the window. On wake up, by measuring the boost over only the first half, the process can be compensated exactly for the slept time.

The sweeping window is implemented as a circular buffer (a shift register) of tuples (t, s, b) recording the time length t of a state, the share s during this time, and a boolean b indicating if the process was running or not during this state. Each time a process has its state changed on schedule operation, the previous state and its duration are recorded by the scheduler in the process history window. The allocated share over the window

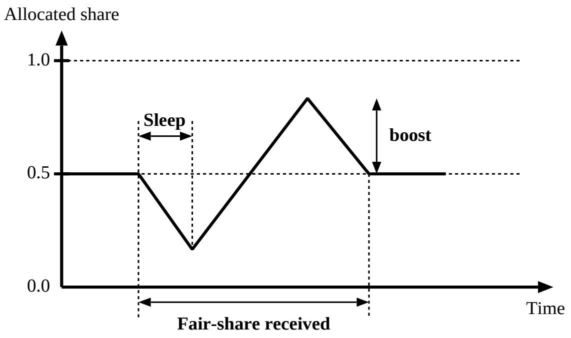

Figure 11: *A process having a long CPU burst and sleeping only very short intervals can be allocated its active share.*

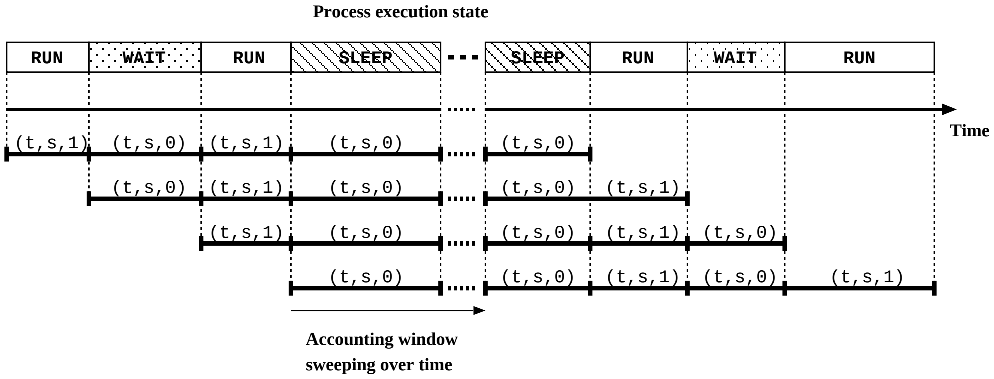

Figure 12: *Recent CPU allocation measurement over a sweeping window.*

is then calculated simply with the following formula.

$$allocated\ share = \frac{\sum t_i * b_i}{window\ time\ length}$$

Where $\sum t_i * b_i$ is the sum of running states time length recorded in the window.

To avoid scanning all the window each time it is desired to evaluate the recently allocated share, the sum of the CPU time received is incrementally modified each time a new tuple is added to the window (that is on event occurrence). Three variables are used to manage the start and the end index of the sections in the array representing the total window and its first half, as shown in figure 14.

When a new information is added, the start index is increased by one, and the end index increased if necessary so that the total time length between the start and the end element do not exceed the specified time length of the window. This total time length is also maintained incrementally each time new information are added and old information erased from the window. The same method is used for the first half window. Doing so, the overhead of maintaining such precise execution history information is kept very small.

The next section presents the scheduling algorithm combining priority and stride scheduling which uses this accounting window to control CPU time allocation.

3.6 Priority and Stride Scheduling Combination

The core idea of the scheduling mechanism implementing the CPU time allocation policy presented previously is to combine a classical priority scheduler with the implementation of stride scheduling presented in section 3.3. Priority scheduling can provide the desired control over process scheduling order, while stride scheduling can be used to implement simply fair-share scheduling without a difficult priority calculation method.

Process preemption on wake up can be controlled efficiently to minimize the response time on interactive processes. To implement the boost of a process on wake up, scheduling can be done by assigning a high priority to this process until it is allocated its active share over the first half of the accounting window. The deterministic behavior of stride scheduling allows to control precisely the "*unboost*" of a process. The overall fairness relies on the use of the total accounting window presented in the previous section.

The ready queue has a classical multilevel feedback

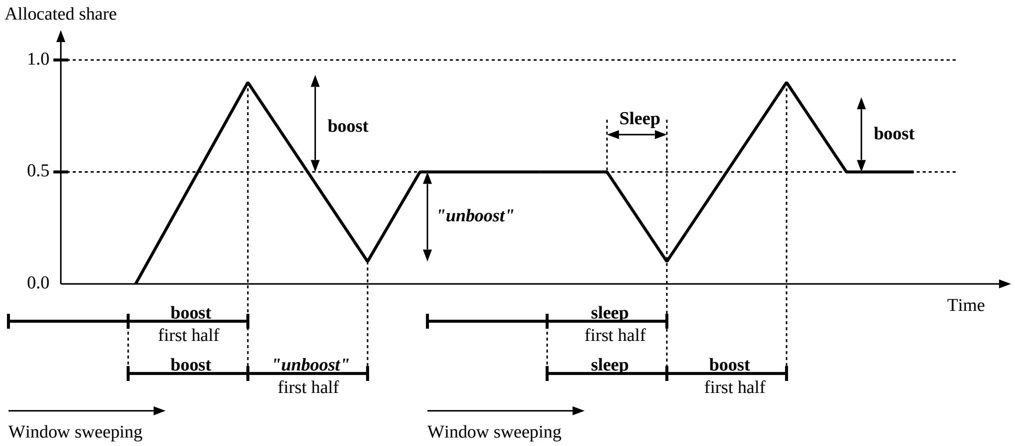

Figure 13: *Sweeping window use.*

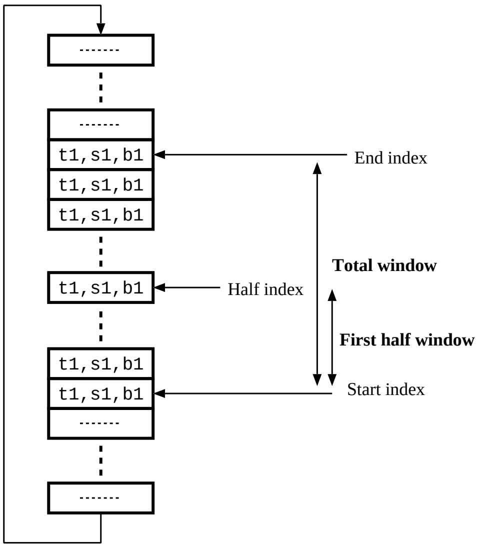

Figure 14: *Accounting window data structure.*

queue structure. Priority scheduling is used between queues of processes of the same priority. As shown in figure 15, the stride scheduling queue presented in figure 8 is used as the process queue of priority level 1. All other priority levels use the RR policy with different time quantum. Processes are chosen in decreasing priority order.

Stride scheduling is thus used only if no process has a higher priority than the stride queue priority. The calculation of priorities and time quantums is explained in the next section. It is followed by the explanation of the control mechanism ensuring fair-share allocation of CPU time using the stride queue. The CPU starvation problem which may occur in some cases is then discussed and a solution proposed.

3.6.1 Priority Calculation

Unlike priority based fair-share schedulers, only a simple priority calculation method is used.

This method gives a high priority and a short time quantum to process releasing voluntarily the CPU often when performing blocking system calls, whereas long CPU burst processes (in particular compute bound processes) have their priority lowered and their time quantum increased. Such classical priority and time quantum calculation scheme allows a good emulation of the SJF policy, minimizing the average wait time in runnable state. New processes are always assigned the highest priority at start time.

Figure 16 shows the table used as a base to calculate priorities and time quantums. For each priority level, this table defines the next priority `priexp` when a pro-

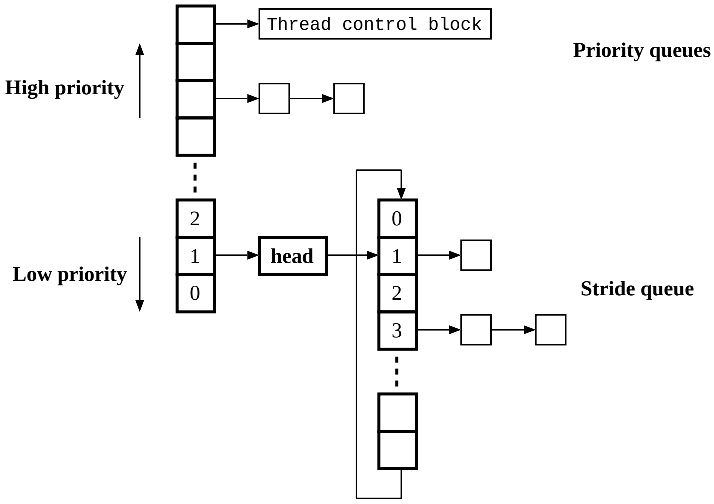

Figure 15: *Ready queue structure.*

cess uses all its time quantum, the priority on wake up `prisslpret` if a process sleeps waiting for an event from this priority level and finally the time quantum `quantum` 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 occupy more priority levels.

The stride queue priority is 1 and a default time quantum of 100 milliseconds is used for this priority level. The priority level 0 is reserved for kernel process execution, when no user processes are runnable. In particular, the idle process is executed using this priority level.

As for share and ticket update, the scheduler recomputes priorities on event occurrence for the processes concerned by the occurring event. For example, when a process is preempted either at the end of its time quantum by the last clock tick or by a higher priority process waking up, the preempted process priority is calculated using the table. A process performing a blocking system call has its priority on wake up set before sleeping.

However, as the next priority field for each priority level is never one, the stride queue is never naturally used by computing priorities using this table. Stride scheduling usage is explained in the next section. It depends on the run-time behavior of a process and in particular on the recently allocated share of CPU time measured over the sweeping window for each process.

3.6.2 Stride Scheduling Usage

The use of the stride queue at the priority level 1 is determined depending on the recent CPU usage measured over the accounting window. It aims at implementing the overall desired fair-share allocation of processing time between all processes.

The scheduler calculates the CPU allocation of a preempted process over the first half of its accounting window before recomputing its priority. The following two cases can occur.

Active share not allocated If the preempted process was allocated less than its active share over the first half of the accounting window, its priority is decreased using the priority table depending only on its current priority, and on its time quantum usage. In this case, its priority is never decreased to 1, as the `priexp` field for each priority level higher than the stride queue priority is never 1. Doing so, this process will still be scheduled using priority scheduling.

Active share allocated On the contrary, if the preempted process was allocated its fair-share or more over the first half of the window, its priority is set to 1 so that it will be scheduled next time using stride scheduling, when no other higher priority process is runnable.

A process will thus be scheduled ahead of processes in the stride queue using priority scheduling until it is allocated its fair-share over the first half of the accounting window. Processes in the stride queue are only those which were allocated at least their active share.

The next section will discuss in more detail the effect of this control mechanism on the CPU time allocated over the total time length of the accounting window. As it may be necessary to penalize a process, the "*unboost*" control is also discussed.

3.6.3 Penalty

As explained previously, a process is scheduled with stride scheduling only if it has been allocated at least

```
int pritbl[] = {

        // PRIO    quantum priexp prilslpret

        /*  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 16: *Priority table.*

its active share over the first half of the window. But as it will be guaranteed to receive exactly its active share with stride scheduling, this may result in an over share allocation. This situation is shown in figure 17.

In this example, a long run compute bound process starts execution. Its active share is 50%, and run concurrently with other long run compute bound job (for simplicity, we do not consider the case where this process is preempted by another one). This process will thus be scheduled with priority scheduling until it is allocated 50% of the CPU time over the first half of the window. As it is scheduled with a high priority ahead of other compute bound processes, it will be able to use 100% of the CPU time over the first quarter of the window, which is 50% over the first half (window 1). It will then be scheduled with stride scheduling and will be allocated exactly its active after (window 2 to 5). Thus over the first complete window (window 4) it is allocated $100\% + 3 * 50\% = 62\%$, which is higher than its active share. In such case, the process should be penalized.

This is done simply by allocating only the CPU time necessary to match the active share over the first complete window, by modifying the time quantum of the process when it is scheduled with stride scheduling. Indeed, considering the deterministic behavior of stride scheduling, the number of times n a process is scheduled over a time T is:

$$n = process\ activeshare * \frac{T}{tq}$$

where tq is the time quantum length used. As the process receives 100% of the cpu time over the first quarter of the window, it should thus be scheduled with a shorter time quantum over the remaining three quarters of the window. For a window time length w, a process should thus be assigned a time quantum tq for n times, calculated as follows if the default time quantum is 100 milliseconds.

$$tq = 100 * \frac{4 * active\ share - 1}{3 * active\ share}$$
$$n = process\ active\ share * \frac{3 * w}{4 * 100}$$

As a process may be preempted by a higher priority process waking up, it may not receive 100% on the first quarter of the window. It also may sleep, and moreover, as the allocated share over the first quarter of the window is not measured, the time quantum is approximated as follows:

$$tq = 100 * 3 * \frac{2 * active\ share - allocated\ share}{4}$$
$$n = process\ active\ share * \frac{3 * w}{4 * 100}$$

where *allocated share* is the maximum share of CPU time allocated over the first half of the window. This approximation gives smaller time quantum than the exact result, a process is thus penalized more quickly. A simple test allows to stop this penalty as soon as the CPU allocation over the total time window becomes fair. This test also allows to not penalize a process

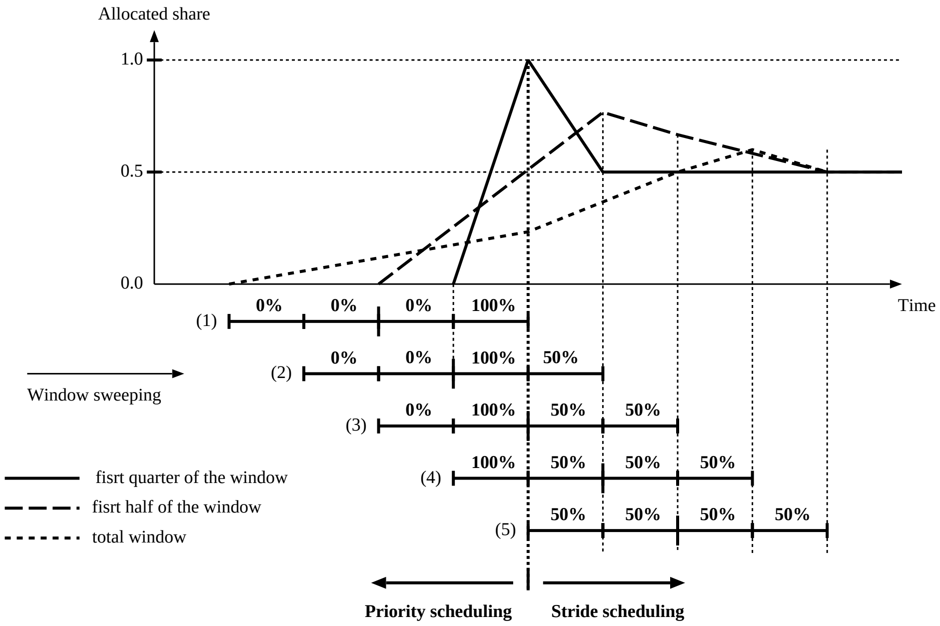

Figure 17: *CPU time allocation measured over the whole or part of the accounting window.*

which was sleeping for a time interval smaller than the window length.

Using this scheme, the CPU starvation problem may appear in some cases. Indeed, a process can be assigned the priority of the stride queue only on preemption at the end of its time quantum or by a higher priority process waking up. As a result, I/O bound or interactive processes always releasing the CPU before the end of their time quantum may be able to use more than their active share, because they will never be penalized. A solution is to add a control point on return from system call, when a process is about to return to the user space. This last control is explained in more detail in the following section.

3.6.4 Average Active Share

A process sleeping waiting for I/O completion has its priority on wake up calculated using only the priority table. This priority is higher than the stride queue priority to reduce the response time of interactive and I/O bound processes. Processes always releasing the CPU before the end of their time quantum will thus never be scheduled with stride scheduling, and may be able to use more processor time than their active share.

Considering only the active share of a process calculated with active tickets can introduce problems to control the CPU time allocation to such processes. Let us consider the example of a user A owning half of the user tickets of a node and running five processes with the same number of tickets. Another User B is running a single long run compute bound process. User A processes are synchronized so that only one process is runnable at a time, with a constant CPU burst of almost 20 millisecond, smaller than the highest priority level time quantum (figure 16). User A should be allocated a total of 50% of the cpu time, in the average 10% for each of his processes.

As only the tickets of runnable or running processes (active tickets) are considered to calculate active shares, each of user A process will always have an active share of 50% on wake up. User A processes will be executed ahead of user B compute bound process, and as they will never be preempted by the clock tick at the end of their time quantum, they will never enter the stride queue. User A processes will thus be allocated over a window of 1 second a share of $(20\ ms * 10)/1000 = 20\%$ each. User A will receive in total 100% of the CPU time. The scheduler will never detect this unfair allocation as each process never consumes more than its active share of 50%.

On the other hand, as shown in figure 18, the average active share of user A processes over the accounting window of one second can be calculated as $(20\ ms * 0.5 * 10)/1000 = 10\%$, as the share of each of user A processes while sleeping is null (their process tickets are not active). Using the average active share over the recent CPU usage window instead of the calculated active share for the CPU burst of each process gives the right CPU allocation to each process, allowing the scheduler to detect the unfair state resulting if user A processes are not preempted.

However, if a process sleeps a long time, its average

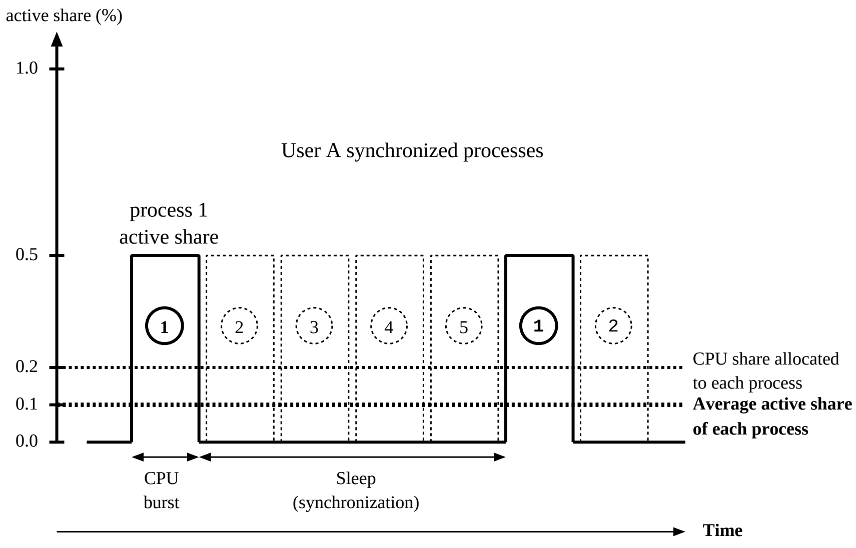

Figure 18: *CPU starvation problem using the active share.*

share may be decreased too much, even become null if ti sleeps during a time interval longer than the accounting window time length. To avoid this situation, the average share is bounded with the minimal share. Indeed, considering the total number of tickets in the system, a process should be allocated at least its minimal share.

The control point added on return to system call allows the scheduler to check if a I/O bound process was not allocated too much CPU time. If the allocated share is higher than the average active share, the process is preempted as soon as it returns to user space, and its priority is decreased to 1, so that the process is next scheduled with stride scheduling. This control 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 next chapter will presents some evaluation results explaining in more detail the effect of the window time length on scheduling. First, results showing the basic behavior of the scheduling algorithm presented are discussed, experiments with various workload and the resulting precision of the CPU time allocation mechanism using the accounting window will then be shown.

4 Evaluation Results

This chapter presents evaluation results and performance measurements of the local scheduler of Colonia. These measurements where done using an emulator developed to test the hardware independent code of the kernel on a classical workstations.

The first section will presents briefly the emulator structure and program execution simulation. Scheduling overhead measurements will be presented next, followed by various experiments results.

4.1 Emulator

The emulator allows to execute the core part of the kernel of Colonia and to simulate process execution using UNIX functionalities. This section will first briefly present the structure of this emulation facility and will discuss program execution simulation used for the scheduler evaluation.

The layer structure of the kernel of Colonia allows to replace the hardware dependent part of the code with an emulation program using classical systems calls of the UNIX operating systems. Higher level code of the kernel can thus be used without modifications by preserving the interface with the hardware dependent code.

Figure 19 shows the general organization and the execution model used to run the emulator. A root process is first started to initialize the kernel data structures. This process can then forks other UNIX processes used to simulate programs execution. Interruptions are emulated using UNIX signals. Context switching is emulated by synchronizing with semaphores Unix processes simulating program execution so that only one is runnable at a time. Each process is able to effectively execute its own user program.

Because of the difficulty of developing such emulator, only few systems calls where implemented. To test the scheduler, each emulated process runs the same program which simulates CPU and I/O bursts. The length of these bursts are calculated randomly. I/O bursts are emulated by calling a system call similar to the UNIX `nanosleep()` system call developed for the emulator.

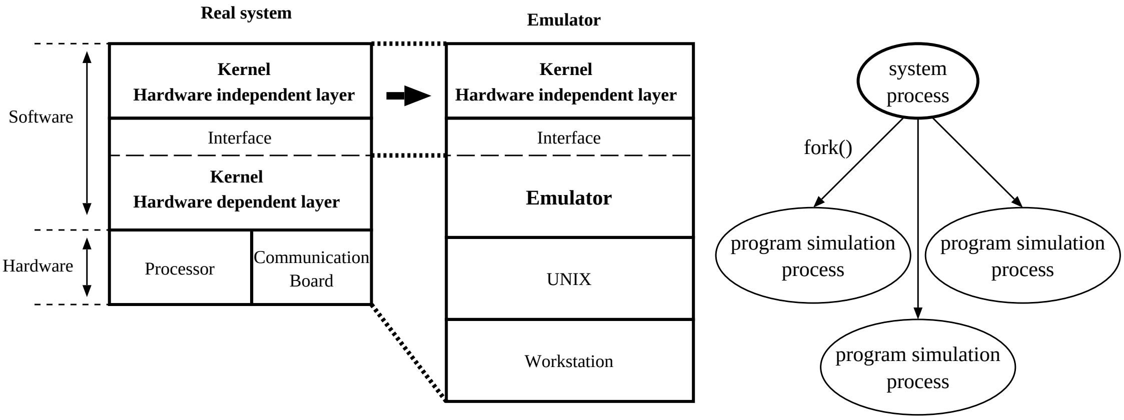

Figure 19: *Colonia emulator.*

The length of CPU and I/O bursts are calculated with an exponential distribution with two averages: one representing the average CPU burst, and the other the average sleep time length. Parameters allow to change dynamically the value of these distribution in order to simulate a change in the run time behavior of a process.

All the results presented in this section were measured using this emulation environment. Overhead measurement are thus "approximated", as some functions using directly hardware features, like time related functions, are emulated.

4.2 Scheduling Overhead

The overhead of queue manipulations for stride scheduling is first presented, measurements of the total overhead of the scheduling method presented previously is then shown.

4.2.1 Queue Manipulation Cost

We have compared the relative cost of stride scheduling queue manipulation for our implementation and for the implementation proposed originally in [10] and presented in section 2.4.3. Figure 20 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, and shows the average number of processor cycles for each function depending on the number of runnable processes in the stride queue.

The get operation chooses a process in the stride queue and removes it. A process entering the stride queue is placed using the put operation. This function calculates the index in the array depending on the previous index, on the stride and on the stride error for our implementation. For the original implementation, pass parameters of processes are updated on put operations. Our implementation provide a bounded execution time for the "put" operation, whereas the use of pass parameters results in a cost in $O(n)$ because of queue sorting. Only a simple sort algorithm was used, a better one would provide a cost in $O(\log n)$. The get operation is slightly slower for our implementation because the head pointer in the stride queue structure needs to be updated each time the head queue becomes empty. This is not needed by the other implementation, resulting in a lower cost.

In the average, the implementation presented in section 3.3 does provide lower queue manipulation cost as expected. The next section present the total overhead of using this structure with stride scheduling.

4.2.2 Overhead

The scheduler uses both priority and stride scheduling. The execution time of the `schedule()` was measured in the case of only priority scheduling or only stride scheduling use. These two cost does in fact bound the average overhead of this function.

When the function `schedule()` is executed, the process running previously is either still runnable (it was preempted) or start sleeping on an event by calling the `sleep(event)` function. `schedule()` updates CPU usage information of the previously running process, update its accounting window with the CPU time received and recalculates its priority depending on its state and on its CPU usage as explained. If the previously running process is still runnable, `schedule()` put it back into the ready queue. If the process is put back into the stride queue its stride is recomputed. `schedule()`. The next process to execute is then chosen, and its accounting window updated with the waited time.

The graph shown in figure 21 represents the execution time of the `schedule()` function in micro seconds depending on the number of runnable processes,

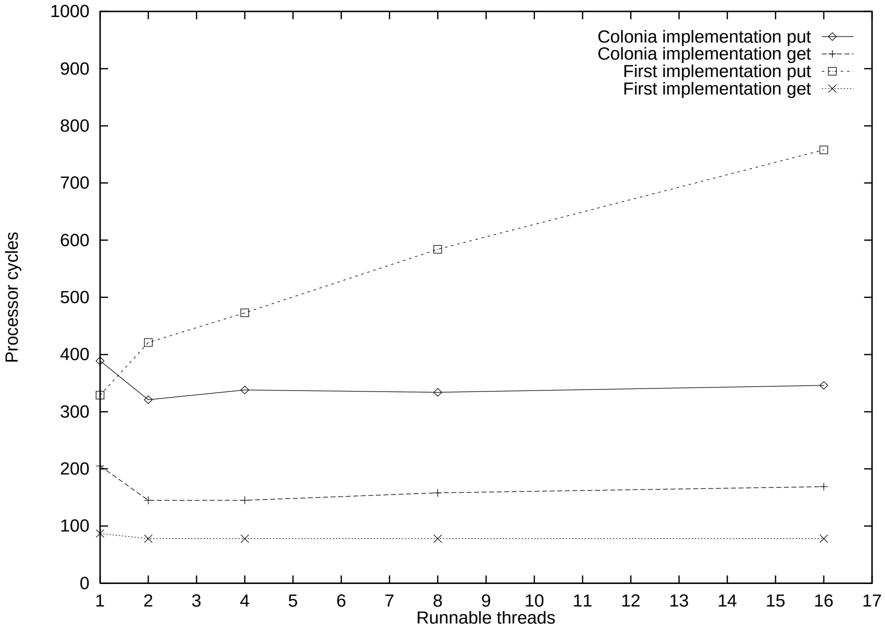

Figure 20: *Stride queue manipulation cost (put and get operations).*

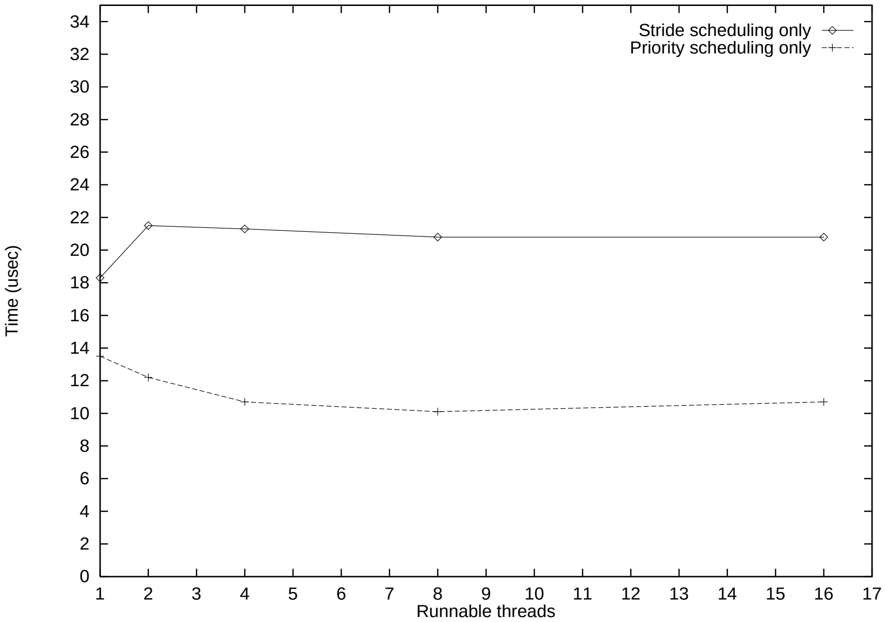

Figure 21: *Scheduling overhead when either priority scheduling or stride scheduling is used.*

when only priority scheduling or only 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 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. On the contrary, priority scheduling overhead will be more representative on lightly loaded nodes.

4.3 Experiment

In this section, some experiments results measured using the program execution simulation of the emulator are presented. 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.

4.3.1 Basic Behavior

This section illustrates the case discussed in section 3.6.3. It shows the effect of the window time length on the scheduler behavior.

In this simple experiment, a user A owning half of the active user tickets of the system is running a single process (process 1) starting execution after 10 seconds. This process has three CPU bursts of respectively 300 milliseconds, 600 milliseconds and 5 seconds. Another user B is running concurrently two long run compute bound processes which never sleep.

Figure 22 shows execution results with a 2 seconds accounting window. The graph shows the allocated CPU time of process 1 measured over an external window of 1 seconds, that is half of the accounting window. The first short CPU burst of process 1 is executed quickly, in fact process 1 release the CPU only after completing it as its share over the first half of the window is less than its active share (30% against 50%). For the second CPU burst of 600 milliseconds, the same behavior is observed, as process 1 should at least be allocated its active share over the first half of the window. The resulted measured share is thus 60% over the first half, process 1 execution was boosted. It was however allocated only 30% over the total window. Finally, as the last CPU burst is longer, process 1 is penalized after its boost, so that the 5 seconds CPU burst is executed in exactly 10 seconds.

The same example is also shown in figure 23 and 24 with respectively accounting window length of 1 and 4 seconds. These figures also shows the allocated share of process 1 measured over a 1 second external window.

For a 4 seconds accounting window, the same behavior as figure 22 can be observed, at the exception of higher amplitude boost and penalty due to the longer accounting window. On the contrary, for a 1 second accounting window case of figure 23, process 1 never uses more than its allocated share. This is also due to the fact that measurements where done on the same time length as the accounting window. Finer observation would show the boost/penalty of process 1.

The execution of the 5 seconds CPU burst of process one is completed in exactly 10 seconds for the three window length presented. The scheduler preserve fairness even if processes are boosted on wake up. However, longer accounting windows degrades the regularity of CPU cycles allocation. This can be a problem to guarantee a constant service rate to processes, shorter accounting window resulting in a more strict fairness may be desirable in this case. This will also be discussed in the next experiment concerning compute bound processes.

4.3.2 Compute Bound Processes

This section illustrates the compensation of the time slept by processes with the boost on wake up. In this experiment, User A is running a compute bound job which in the average has CPU bursts of 700 milliseconds and I/O bursts of 100 milliseconds. User B is running two long run compute bound jobs, with a ticket ratio of 3 to 2. User A and B are allocated the same number of tickets. In total, user A process needs 20 seconds of CPU time to complete, and sleeps 1.89 seconds over the entire run.

Figure 25 shows the execution results measured over a one second external window, but the accounting window used was 2 seconds. When user A starts his process, his tickets become active, resulting in the readjustment of user B processes share. As process 1 sleeps, the boost on wake up are not penalized if the slept time is exactly compensated. Table 1 shows execution time results (turnaround time) of process 1 and the corresponding CPU time allocation error with different active share.

As process 1 sleeps only small intervals, the CPU time allocation error is small for all time window length, but slightly increases with the active share and with the accounting window length. This is due to the fact that the longer the window is, the more the scheduler will try to compensate process 1 for slept time, resulting in a relative unprecision. If the active share increases, boosting the process execution high above its active share to compensate slept time becomes difficult, resulting also in a slight increase in CPU time allocation error.

In figure 26, shows the scheduler fairness depending on the average time slept by process 1 with the use of different accounting window time length. The fairness

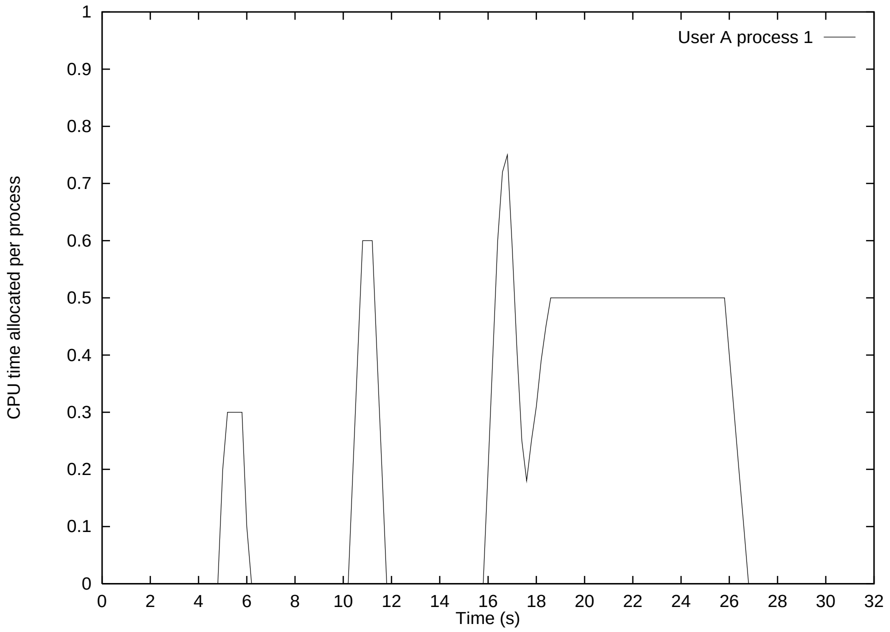

Figure 22: *Basic behavior : accounting window of 2 seconds.*

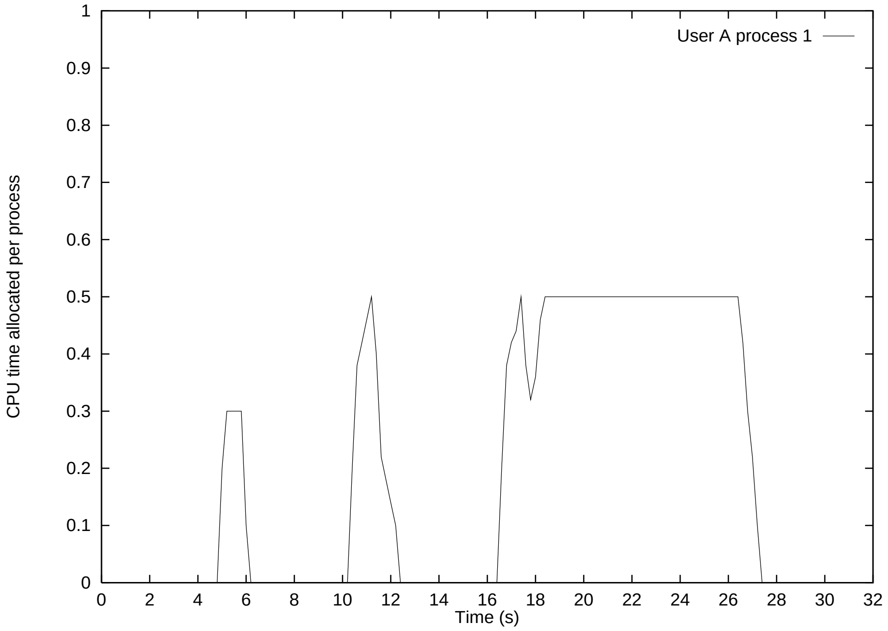

Figure 23: *Basic behavior : accounting window of 1 second.*

Table 1: *Total execution time of process 1 for different accounting window time length and three different active share.*

Window length	Execution time share 25%	Execution time share 50%	Execution time share 75%
1	83.089 (-3.71%)	40.459 (-1.13%)	27.960 (-4.62%)
2	79.919 (+0.10%)	40.179 (-0.44%)	27.389 (-2.64%)
4	80.189 (-0.23%)	39.769 (+0.58%)	26.909 (-0.9%)
6	79.769 (+0.29%)	39.999 (+0.00%)	27.059 (-1.45%)
8	78.379 (+2.06%)	39.259 (+1.88%)	26.699 (-0.12%)

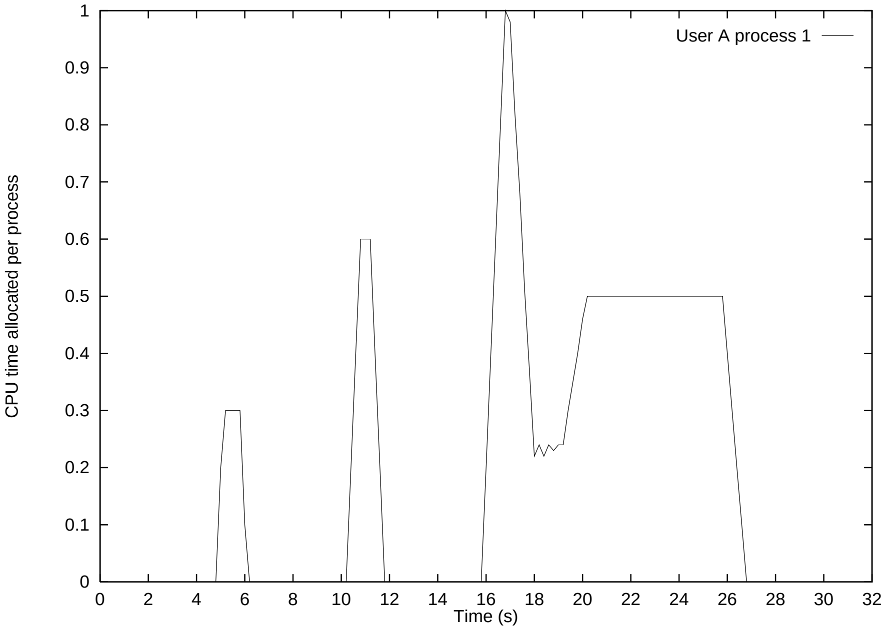

Figure 24: *Basic behavior : accounting window of 4 seconds.*

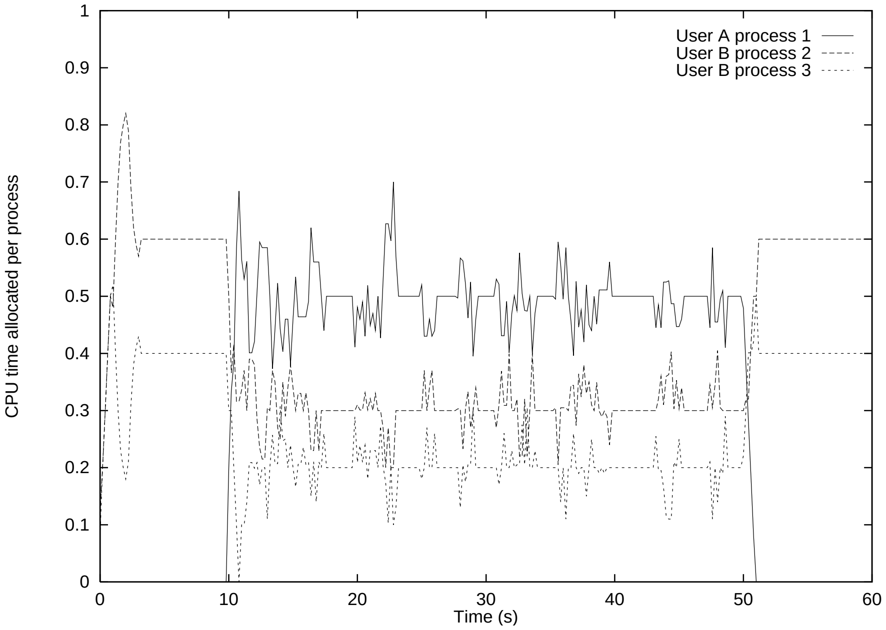

Figure 25: *Compute bound process execution with an accounting window of 2 seconds.*

is simply measured using the following ratio.

$$\frac{active\ share * turnaround\ time}{CPU\ time\ used}$$

That is in this case (active share of 50% and CPU time of 20 seconds) the ratio *turnaround time*/40. If this ratio is 1, the scheduler is fair, if it is smaller than 1, the scheduler has allocated too much CPU time to process 1. On the contrary, if it is higher than 1, process 1 was allocated less than its fair-share of CPU time.

As expected, longer accounting window perform better when process 1 sleeps more, due to the possibility of longer boosts which can compensate longer sleep time intervals. However, the fairness is quickly degraded above 40% of sleep time, because CPU bursts become too short to allow sufficient boosts on wake up to compensate the time slept. However, the scheduler performs well for compute bound processes having long CPU bursts and only short I/O bursts. Fairness is improved compared to strict fair-share scheduling with the use of long enough accounting windows. The behavior of the scheduler can be degraded to strict fair-share scheduling by reducing the size of the window, allowing flexibility in the use of this scheduling method.

The next section considers I/O bound processes, that is processes which generally in the average have I/O bursts longer than CPU bursts.

4.3.3 I/O Bound Processes

In the example of figure 27, three users are running processes of various types. User A with a share of 30% executes seven I/O bound processes with the same number of tickets. User B with a share of 50% executes also seven I/O intensive processes. User C computes with a share of 20% and runs a single long run compute bound process.

The accounting window used is 4 seconds, and the graph presented shows results measured over the same time length. In this example, all users total share allocation oscillates around their active share value as processes sleeps a lot. As almost all the interactive processes in this example are not naturally scheduled with stride scheduling because having too short CPU bursts, the control point added on return from system calls is highly used. The scheduler thus preempt those I/O bound processes which uses more than their average active share to implement global fairness.

Table 2 shows the CPU time allocation for each user and the relative error. Even in the case of a high interactive load, CPU time allocation stays very precise. Different time windows gives similar results, the error only slightly increases with the length of the accounting window.

Not only the allocation is fair for users, but the CPU time allocated to each process of user A and B is also fair. Figure 28 shows this allocation for each of the seven processes of user A and B.

As users A and B are running a lot of processes, each process average active share is low. It is thus easier to compensate the time slept by each process in this case than in the case discussed in section 4.3.2 for figure 26. Each process of user A and B can thus be easily allocated its fair-share.

This section has shown that the use of the average active share over the accounting window can provide a great precision for CPU time allocation to I/O intensive processes. Even in the case of a mixed load of compute bound and I/O bound applications the scheduler is able to allocate to necessary CPU time to match the share of users and of their processes.

5 Conclusion

In order to both provide a fair allocation of CPU time to share efficiently processing time between competing users and processes, the scheduling method presented enhance the basic stride scheduling method with the use of a classical priority scheduler. Priority allows to implement a more precise control over process scheduling order than a dynamic change in ticket allocation. Moreover, priorities provide the system with a mechanism to simply implement process preemption, so that I/O bound and interactive processes waking up can be scheduled sooner, minimizing their response time to event occurrence.

The scheduling algorithm boosts the CPU time allocation of processes on wake up, in order to allocate to a process its fair-share of CPU time quickly. This allocation policy combined with a classical priority calculation method allows to simply emulate the short-job-first policy. A process having a short CPU burst may be able to complete it faster than with a strict fair-share scheduling method.

The overall implementation of fairness relies on the use of a precise mechanism to measure CPU time allocation, based on a sweeping window. This mechanism controls the use of stride scheduling by determining when a process should enter the stride queue. The deterministic behavior of stride scheduling is also used to dynamically adjust the time quantum of processes which were allocated more than their fair-share. This mechanism results in the use of stride scheduling only for processes which are using all their fair-share of cpu time. As I/O bound and interactive processes have generally short CPU burst alternated with long I/O bursts, on lightly loaded nodes, the behavior of the scheduler is automatically degraded to use only priority scheduling. On the contrary, on more heavily loaded systems, fair-share allocation of CPU time is possible while pre-

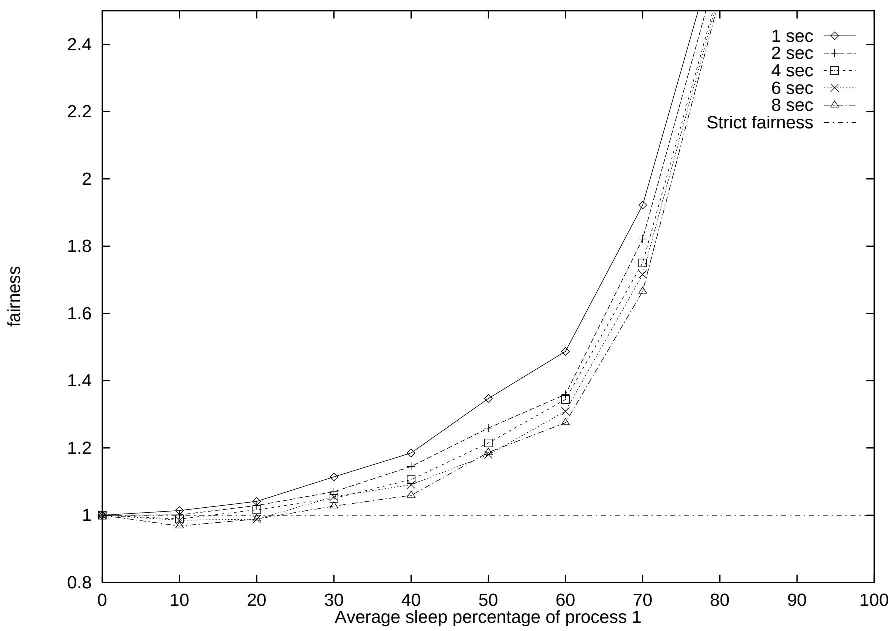

Figure 26: *Scheduler fairness depending on the average sleeping time of process 1 of the exemple of figure 25.*

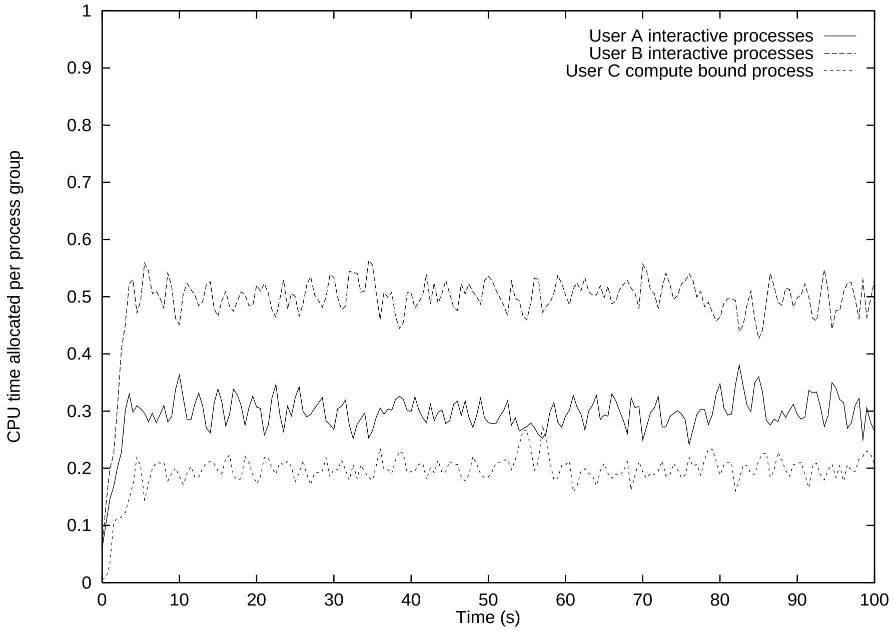

Figure 27: *Mix of load with a high I/O bound process workload.*

Table 2: *CPU time allocation for each user over the total time length of the example of figure 27.*

User	Allocated CPU time	Error
A	29.994	-0.02%
B	50.084	0.17%
C	19.922	-0.39%

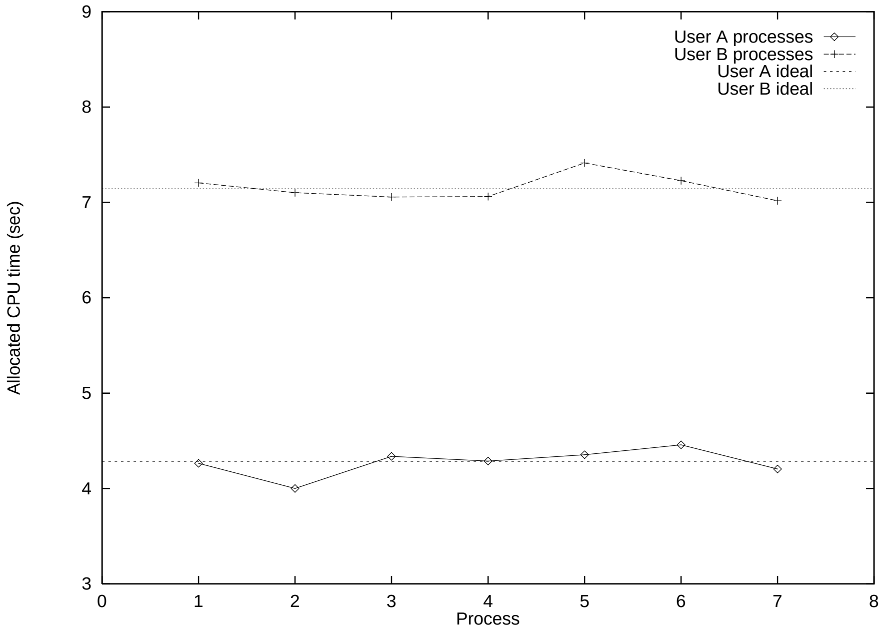

Figure 28: *CPU time allocated per process for user A and B of the example of figure 27.*

serving the response time of interactive processes.

Evaluation results have shown the precision of this scheduling method for various workload. Not only for each user, but also for each of their processes, the scheduler allocates precisely CPU time to match active shares of users and processes. Event-based scheduling allows to keep data management simple, and to reduce scheduling overhead by considering only few processes at a time. It has been also shown that the implementation of stride scheduling proposed in this thesis allows to reduce scheduling overhead, and to suppress the dependence of this overhead to the number of runnable processes.

The use of the accounting window time length as a parameter can change the relative strength or weakness of the overall fairness by allowing a process having a long CPU burst to be compensated for the past slept time, resulting in improvement of CPU time allocation over strict fair-share scheduling. However, this control has a not negligible effect on the distribution of CPU cycles among processes. Longer accounting windows improve fairness for processes sleeping on event occurrence, but does results in unstable allocation of CPU time.

Future work should investigate more thoroughly the effect of the boost/penalty combination controlled with the accounting window on the service rate provided to processes. Indeed, Quality of Service considerations would suggest to be able to provide a more stable distribution of CPU cycles to processes and to be able to control it more precisely while preserving the improvements of this scheduling method. Implementation on an operational system would also allow a more precise tuning by allowing to test the scheduler behavior against "real" load.

Acknowledgments

I would like to express my sincere gratitude to Professor Shinji Tomita who gave me the chance of participating in this research project, and for his help and support in my daily life as a foreigner in Japan. I would like also to thanks Professor Shin-Ichiro Mori and Yasuhiko Nakashima for their helpful comments and discussions during the development of the work presented in this thesis.

I would like to express particular thanks to Research Associates Masahiro Goshima for giving me his continuous assistance during this research and for his many helpful comments on the presentation of this work. I would like also to acknowledge Mister Shin-Ya Date, Mister Mineyoshi Masuda and all the students who have contributed with me to the Computer Colony project.

Finally, thanks are also due to the Ministry of Education, Science, Sports and Culture of Japan who granted me a scholarship during all my studies in Japan.

References

[1] Damien Le Moal, Mineyoshi Masuda, Shin-Ya Date, Hiroshi Yamazoe, Masahiro Goshima, Shin-Ichiro Mori, Shinji Tomita "A Fair-Share Prior-

ity Scheduler for a Distributed Operating System", Summer Workshop on Parallel Processing (SWOPP), 99-OS-82, page 49-56, 1999.

[2] S. Khanna, M. Sebree, J. Zolnowsky "Real-time Scheduling in SunOS 5.0", USENIX Conference Proceedings, Winter 1992.

[3] AT&T Unix System V Release 4 Programmers Guide "System services and Application Packaging tools", Prentice Hall, 1990.

[4] Prabhat K. Andleigh "Unix System Architecture", Prentice Hall, 1990.

[5] Marshall Kirk McKusick "The Design and Implementation of the 4.4BSD Unix Operating System", Addison Wesley, 1996.

[6] J. Kay, P. Lauder "A Fair Share Scheduler", Communications of the ACM, Volume 31, Number 1, January 1988.

[7] Raymond B. Essick "An Event-based Fair Share Scheduler", USENIX Conference Proceedings (Winter 1990).

[8] 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.

[9] David Petrou, John W. Mildford, Garth A. Gibson "Implementing Lottery Scheduling : Matching the Specializations in Traditionnal Schedulers", Usenix Annual Technical Conference (http://www.usenix.org/events/usenix99/technical.html), June 1999.

[10] 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.

[11] Andrea C. Arpaci-Dusseau, David E. Culler "Extending Proportional-Share Scheduling to a Network of Workstations", International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) Proceedings, 1997.

[12] A. Goscinski "Distributed operating systems - the logical design", Addison Wesley, 1991.

[13] Jean Bacon "Concurrent Systems - Operating Systems, Database and Distributed Systems : an integrated approach", Second edition, Addison Wesley, 1997.

[14] A. Silberschatz, J. Peterson, P. Galvin "Operating System Concepts", Third edition, Addison Wesley, 1991.