

Round Robin with Sharetokens: a CPU Scheduler for Heterogeneous Application Mixes

Oveeyen Moonian and Geoff Coulson

Distributed Multimedia Research Group

Lancaster University

Lancaster

+44 1524 593054

ovm@uom.ac.mu, geoff@comp.lancs.ac.uk

ABSTRACT

General purpose workstations must support a wide variety of application types, but it is hard to find a single CPU scheduling scheme that can satisfactorily schedule processes from all types of application. In particular, it is difficult to get deadline-driven continuous media processes to satisfactorily co-exist with others. A number of schemes have been developed to address this issue, but these are often either inefficient, non-deterministic, unable to explicitly support deadlines or unable to exploit the adaptive potential of continuous media applications. This paper presents Round Robin with Sharetokens—a low-overhead, deterministic, adaptive, proportional-share scheduling scheme that enables deadline-driven processes to meet their deadlines while ensuring that others obtain their fair share of CPU time.

1. INTRODUCTION

Modern workstations and PCs need to support a heterogeneous mix of applications including interactive, computation intensive, and multimedia types. Unfortunately, this variety is ill-served by currently-applied CPU scheduling techniques. In particular, *continuous media* applications have (soft) real-time requirements that are poorly handled by today's general-purpose schedulers.

Typically, today's OSs deal with continuous media applications either by giving them higher priorities or by implementing a 'real-time' scheduler on top of a standard time-sharing scheduler. Unfortunately, the first approach (exemplified by Linux) tends to unduly penalise non-real time applications while the second (exemplified by UNIX SVR4) can additionally lead to unpredictable behaviour on overload, allowing runaway real-time activities to cause basic system services to lock up and the user to lose control over the machine [Nieh 93].

As a result, significant recent research (e.g. [Waldspurger 94 & 94b], [Goyal 96] [Ford 96], [Nieh 97 & 01], [Bruno 98]) has been carried out on new scheduling approaches that attempt either to allocate CPU shares proportionately to different processes or to explicitly support the needs of heterogeneous application types. Unfortunately, as discussed in section 6, all or most of this research suffers for one or more of the following deficiencies: they either incur high overhead, are non-deterministic (and thus unable to adequately support real-time processes), do not support deadlines (and thus do not adequately support continuous media), or do not exploit the *adaptive* capabilities of continuous media applications [Northcutt 91].

In this paper we present a novel approach to supporting continuous media applications in a workstation environment that

suffers from none of these deficiencies. *Round robin with sharetokens* (RRWS) is a simple scheme that combines elements of stride scheduling [Waldspurger 94b], weighted round robin [Silberschatz 98], and earliest deadline first [Liu 73] approaches to yield a deterministic proportional share scheduler that also supports deadline-driven processes.

The remainder of the paper is structured as follows. Section 2 describes the scheduling algorithm and section 3 then analyses the way in which the scheduler treats deadline-driven periodic processes. Following this, section 4 discusses our implementation in Linux and also presents our approach to continuous media adaptation (i.e. policing application and notifying them of altered CPU shares). Then section 5 offers an experimental evaluation of our implemented scheduler and section 6 compares our design with related research. Finally, we present concluding remarks in section 7.

2. ROUND ROBIN WITH SHARETOKENS

2.1 The Basic Algorithm

RRWS is presented incrementally in this and the following subsections. Initially, we make the constraining assumption that processes do not block and then relax that assumption in the subsequent subsections.

On process creation, each process is allocated a number of *sharetokens* to represent its required proportion of CPU time. The proportion of CPU time allocated to a given process P is the ratio of P's sharetoken count to the total count of sharetokens held in the system (i.e. by P and all other processes). When a process is created, it is allocated a single sharetoken by default although this allocation can subsequently be changed.

All sharetokens are held in a central *sharetoken queue* managed by the scheduler. The x sharetokens (where $x \geq 1$) held by each process are distributed in as evenly spaced manner as possible in the queue (see figure 1 which shows the $x=2$ sharetokens owned by a newly created process P_3 being added to an existing queue configuration). More precisely, assuming there are already n sharetokens in the queue when a new process is added, we calculate $spacing = (n + x) / x$ and then place the sharetokens at multiples of *spacing*, but rounded. For example, if a process with 5 sharetokens is added to a queue with 8 sharetokens, *spacing* will be $(8 + 3) / 3 = 2.6$, and the 3 new sharetokens will be placed at positions 3, 5, 8, 10, and 13.

Additionally, the first of the x sharetokens is placed at some random point in the queue (e.g. as determined by the current queue pointer position). This helps ensure that the sharetokens

from each process automatically remain evenly spaced as the queue grows and shrinks as processes are added and deleted from the system.

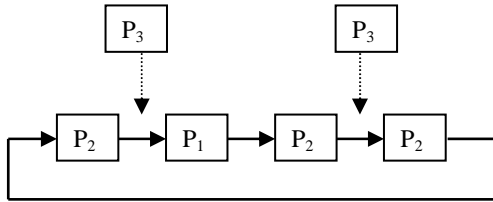


Figure 1: An example sharetoken queue.

Given this arrangement, scheduling is performed by successively allocating CPU quanta to successive sharetokens in the sharetoken queue in a cyclic, round robin, fashion. Thus, after one cycle through the queue each sharetoken has received one quantum on behalf of its owning process. Hence a process owning x sharetokens will obtain the CPU for x quanta per cycle. In figure 1, for example, processes P_1 , P_2 and P_3 hold one, three and two sharetokens respectively, and therefore in each cycle respectively obtain that number of quanta.

2.2 Dealing with Blocking Processes

The above algorithm ensures proportional share over whole numbers of cycles, assuming all processes are runnable. However, if a process *blocks*, its current quantum may not have been fully consumed. Furthermore, some sharetokens will belong to processes that may be already blocked when their turns come. Such processes will therefore not obtain their full CPU allocation (because in such cases the scheduler would simply skip to the next sharetoken that has a currently runnable owner).

In order not to penalise such processes, we introduce a *waking queue* into the basic scheme after the fashion of ‘virtual round robin’ [Haldar 91]. Whenever a blocked process wakes up, a ‘temporary’ sharetoken is allocated for it on this queue. If the process has an associated deadline (see below), the sharetoken is inserted into the queue in earliest deadline first (EDF) order. Otherwise, the sharetoken is simply appended to the back of the queue.

Having introduced the waking queue, the basic scheduling algorithm is modified to first look in the waking queue for the ‘next’ sharetoken before proceeding to examine the sharetoken queue (this latter is performed as before—except that we simply disregard sharetokens belonging to blocked processes). If it finds the waking queue non-empty, the scheduler gives the CPU to the process that owns the sharetoken at the front of the waking queue. Rather than a standard quantum, the process is given the unused time from the quantum in which it previously blocked. When the process has exhausted this quantum its temporary sharetoken is deleted from the waking queue.

This scheme is adequate for cases in which the waking process has not missed any of its sharetoken-turns while it was blocked. If it has, additional steps are taken as discussed next.

2.3 Dealing with Sharetoken-Turns Lost by Blocked Processes

If a waking process has missed one or more whole sharetoken-turns while it was blocked, it is allowed, within limits, to

accumulate missed CPU time as follows:

- accumulated time is defined as the standard quantum length multiplied by the number of times the waking process has had a sharetoken skipped;
- to recover accumulated time, a fraction of this is added to each of the process’s subsequent quanta until the process has caught up.

To make this scheme workable, it needs to be configured with appropriate *accumulation bound* and *quantum bound* values. ‘Accumulation bound’ refers to the maximum amount of CPU time a process may accumulate. Although an accumulating process is not taking more than its permitted CPU share (it is merely recovering what it already lost), allowing a process to accumulate time indefinitely may have unduly adverse effects on other processes when the accumulated time is being recovered. This is particularly true where admission testing is in force (see section 3.4). ‘Quantum bound’ then determines the maximum quantum length that can be employed in recovering accumulated time. Large quantum bounds will have a relatively larger impact but over a shorter period (and vice versa).

Overall, while the waking queue enables deadline-driven processes to meet their deadlines, and I/O bound processes to recover transient losses of CPU time, the role of the accumulated time mechanism is to promote long-term fairness.

2.4 Complexity Analysis

Adding processes to the sharetoken queue is an $O(n)$ operation (where n is the number of sharetokens in the queue), while deleting processes is $O(m)$ (where m is the number of sharetokens won by the process being deleted). However, these operations only occur when processes are themselves being created or deleted and thus will be very rare in comparison to scheduling decisions. As sharetokens are not removed from the sharetoken queue when a process blocks, there is *no* RRWS-specific overhead for blocking/ unblocking.

Scheduling from the sharetoken queue involves either a continuation of the current sharetoken or moving on to the next sharetoken belonging to a runnable process. Thus the next process to execute is selected in $O(1)$ in the best case—i.e. when the next sharetoken belongs to a runnable process. The scheduler has a worst-case complexity of $O(n)$ (where n is the number of sharetokens in the queue) but this occurs only in the very rare circumstance that only one process is runnable and that process owns a single sharetoken that is immediately behind the ‘next sharetoken’ pointer. Note that under such circumstances the CPU will be operating well below its expected load and can therefore probably well afford the additional overhead. RRWS thus has the desirable property that its overhead reduces as the system becomes more heavily loaded¹.

¹ Note also that would be possible to derive a functionally equivalent variant of RRWS in which sharetokens were removed (or, more accurately, temporarily bypassed by the main queue pointers) from the sharetoken queue while their owning process was blocked. This would yield worse case $O(1)$ scheduling under all loading conditions; however, it would mean that blocking and unblocking became $O(n)$ (where n is the number of sharetokens held by the blocking/ unblocking process—

Scheduling from the waking queue always occurs in $O(1)$ time, since the process at the front of the queue is always the one selected. Adding a non-deadline-driven process to the waking queue is also accomplished in $O(1)$ time. The only $O(n)$ operation (where n is the number of sharetokens in the waking queue) that is likely to occur frequently is the addition of a deadline-driven process to the waking queue. But this will only occur frequently if the system is running a lot of deadline-driven processes, in which case it is arguably a reasonable price to pay for ensuring that such processes meet their deadlines. It is worthwhile emphasising that this overhead is not present at all when the application mix does not include deadline-driven processes.

3. ACCOMMODATING DEADLINE-DRIVEN PERIODIC PROCESSES

In this section, we analyse our scheduler's treatment of deadline-driven periodic processes. First we analyse the behaviour of these processes in an underload situation and then extend the analysis to the overload case.

3.1 Guarantees for Deadline-driven Periodic Processes in Underload

It is straightforward and intuitive to see that RRWS gives 'conventional' processes their proportional share in underload, but it is not necessarily intuitive to see that periodic deadline-driven processes necessarily meet their deadlines in each period.

To see that this is indeed the case, consider a periodic deadline-driven process, P , that requires a processing time of C in every period T . Given this requirement, P will require x sharetokens such that $x/n = C/T$, where n is the total number of sharetokens in the queue following P 's allocation. Consider further that each sharetoken obtains the CPU for a time quantum of q ms. If the system is not overloaded, the following argument demonstrates that P will meet its deadline in each period T .

The argument comprises two cases as follows:

Case I: *When P obtains a sharetoken-turn it 'works ahead'—i.e. it starts work on the next period's computation if it finishes the current one before the quantum has completed.*

Here, when P 's first sharetoken obtains the CPU it will be able to execute for $N = q/T$ periods. However, during this quantum it will actually require only NC ms so it will be able to work ahead for $(T - C)N$ ms. Now, P will not obtain its next quantum until after $(n/x) - 1$ non- P sharetokens, i.e. $((n/x) - 1)q$ ms, have elapsed. During that interval it would nominally require $((n/x) - 1)NC$ ms of processing time. Now, $((n/x) - 1)NC = ((T/C) - 1)NC = (T - C)N$, which is precisely the time that P was able to work ahead in the first quantum. Therefore we can conclude that P will be able to meet its deadlines in the work ahead case.

Case II: *P blocks on finishing its current computation and wakes up at the beginning of its next period.*

Here, when the scheduler encounters P 's first sharetoken, P will use C ms of processing time and then block, leaving CPU

i.e. *not* the total number of sharetokens in the system). We intend to empirically evaluate this variant in our future work.

capacity that it can later recover in the waking queue. Let N equal the number of P 's periods that fit between the start of P 's first sharetoken and the start of its next sharetoken; since these points are n/x sharetokens apart, we can deduce the fact that $NT = (n/x)q$.

We can also see that during NT , P will need NC ms of processing time and that P will therefore have enough time in the waking queue if and only if $NC \leq q$. Taking together this latter fact and the fact (from above) that $NT = (n/x)q$, we infer that P will have enough processing time in the waking queue provided that $C/T \leq (x/n)$, which is necessarily the case in underload situations. Therefore we can guarantee that P will have sufficient processing time in the blocking case. Furthermore, we know that P will obtain the CPU in time for each period because EDF (as used in the waking queue) guarantees that deadlines will be met in underload [Liu 73].

As these two cases exhaust the possibilities, we can conclude that in underload conditions, periodic deadline-driven processes will meet their deadlines.

3.2 Degradation of Deadline-driven Periodic Processes in Overload

We can characterise overload as a situation in which total CPU bandwidth is represented by N notional shares, with 1 sharetoken representing 1 share, and more than N sharetokens have been allocated. Given this, it is easy to see that conventional processes will degrade gracefully in overload—if such a process was initially allocated x sharetokens to represent an $(x/N)\%$ share of the CPU, it will obtain $x/(N + y)\%$ when the system is overloaded by y sharetokens.

However, it is less intuitive to see how a deadline-driven periodic process will degrade in overload. In fact, although its CPU share will certainly degrade gracefully, it may not be the case that the number of deadlines met by such a process will degrade proportionately. To see this, consider the following scenario:

- a system represents the CPU resource with 12 sharetokens and employs a standard quantum length of 200ms;
- two deadline-driven periodic video processes, P_1 and P_2 , are created, each of which has a period of 60 ms;
- P_1 has a per-period processing requirement of 40 ms (i.e. it needs $40/60 = 66.66\%$ of the CPU), while P_2 has a per-period requirement of 25 ms (i.e. it needs $25/60 = 41.66\%$ of the CPU);
- P_1 starts at time 0 and P_2 starts just after P_1 so that the period for P_2 will always occur just after that of P_1 ;
- if a process is still processing a frame when its next period starts it will have to drop that frame.

Under this scenario, P_1 would require $66.66 * 12 = 8$ sharetokens, and P_2 would require $41.66 * 12 = 5$. Evidently, this is an overload situation; assuming P_1 was created first, P_2 would actually obtain $5/(5 + 8) = 38.5\%$ of the CPU, leaving P_1 with the remaining 61.5%. The sharetoken queue would then appear as follows: $P_1, P_1, P_2, P_1, P_2, P_1, P_1, P_2, P_1, P_2, P_1, P_1, P_2$.

Now, consider the behaviour of this scenario over its first 1000ms

of operation (after this point its behaviour simply repeats). Over the first 600 ms (i.e. 10 process-periods) P_1 will obtain 400ms (two sharetokens-worth) of processing time and P_2 will obtain 200ms (one sharetokens-worth). Over each of these 10 periods, as P_1 will have a slightly earlier deadline than P_2 , P_1 will execute for its full 40 ms. But each time P_2 obtains the CPU, it will only be about 20 ms from its deadline; thus if it carries on processing, it will only waste the processing time since it will surely miss its deadline. Over the next 400ms, there will be between 6 and 7 periods. In the first 5 of these P_1 will monopolise the CPU as above: at the end of these 5 periods, P_1 would have again met 5 deadlines, but P_2 would have used up 100 ms without meeting any deadline. In the 6th period P_2 will obtain processing time and meet its deadline, whereas P_1 will not. But after P_2 's 25 ms it will block thus giving P_1 its turn again, at which point the cycle repeats.

We can therefore see that although the proportional share of CPU time degrades gracefully under overload, the performance of some deadline-driven processes may suffer unduly—in the case above P_1 meets almost all of its deadlines while P_2 meets almost none.

To address this situation, we could arrange for deadline-driven periodic processes in the waking queue to be given only their allocated CPU ratio in each period. Hence, if it is in the waking queue, a deadline-driven periodic process holding $x\%$ of the CPU would be blocked as soon as it had used $x\%$ of its current period. This should force processes to readjust their processing according to CPU share available. Relating this to the above scenario, despite P_1 having an earlier deadline, it would only be able to use 36.9 ms out of the 60, leaving 23.1, which is the allocated share of P_2 . Provided both P_1 and P_2 can adjust their processing to the CPU share they actually receive, they would be able to carry on usefully in the face of overload. At the moment, however, the details of this possible extension to RRWS remain an issue for future work.

4. IMPLEMENTATION

Our prototype scheduler has been implemented under Red Hat Linux version 7.2, kernel version 2.4.2.

4.1 Kernel Data Structures and Functions

As required by RRWS, our implementation includes one sharetoken queue and one waking queue. In addition, we have added the following fields to Linux's *task* data structure:

- *int no_of_sharetokens* represents the number of sharetokens allocated to the task (hereafter referred to as 'process');
- *int accumulated* represents the CPU time accumulated by the process due to lost sharetoken-turns;
- *boolean is_deadline_type* denotes whether or not the process should be scheduled according to deadlines;
- *unsigned long deadline* represents the next deadline (only used if *is_deadline_type* is true).

Each newly created process is allocated one sharetoken by default (more can be added later as described below), and when the system boots the sharetoken queue is initialised with a single sharetoken for *init_task*. To allocate sharetokens, the *fork()* system call has been modified to set the *task* structure's

no_of_sharetokens field to 1, and to call a function *addtosharetokenq()* to add a sharetoken to the sharetoken queue. There is also a function *delfromsharetokenq()* to delete sharetokens from the queue, and analogous *addtopriorq()* and *delfrompriorq()* calls to add/ remove sharetokens to/ from the waking queue. *Addtopriorq()* is called by Linux's *wakeup()* function so that a temporary sharetoken is added to the waking queue for every waking process. The *delfrompriorq()* function is called by the scheduler when it has scheduled a temporary sharetoken from the waking queue.

4.2 Scheduler Implementation

If the waking queue is not empty, the scheduler schedules the process that owns the first sharetoken in the waking queue, as long as it is runnable and its time counter (i.e. the time unused from the quantum in which it last blocked) is greater than zero. If process is not runnable or the time counter is zero, (this latter can occur when a process wakes up, performs some processing, and then blocks again), the scheduler simply deletes the sharetoken from the waking queue and considers the next one. This continues until the scheduler either finds a process to schedule from the waking queue or the latter is empty.

If the waking queue is empty then the scheduler considers the sharetoken queue. If it is not the case that the process owning the "current" sharetoken on the queue is runnable and has not already exhausted its quantum, the scheduler continues by scanning the queue until it finds such a process.

Exiting processes are dealt with by removing their sharetokens from the sharetoken queue before placing them in the 'zombie' state.

Regarding the *accumulation bound* and *quantum bound* scheduling parameters, we currently specify and implement these as follows: *i)* we set quantum bound for all processes to twice the length of a standard quantum, and *ii)* we set accumulation bound on a per-process basis to quantum bound multiplied by the number of sharetokens owned by the process. We then distribute any accumulated time equally over each of the process's sharetokens. We plan to experiment with the effects of varying this simple scheme in our ongoing work.

4.3 System Call API

The following new system calls enable control over the scheduler:

- *set_sharetokens(int pid, int numsharetokens)* modifies the specified process's sharetoken allocation;
- *get_sharetokens(int pid)* returns the number of sharetokens allocated to process *pid* or, if *pid = 0*, the total number of sharetokens allocated by the system (not counting the waking queue's temporary sharetokens);
- *set_deadlinetype(int type)* sets the calling process to be either a deadline-driven process or a conventional process (the latter is the default on creation);
- *set_deadline(unsigned long next_deadline)* is used to set the next deadline of the calling process (which is assumed to be a deadline-driven process) to the current time plus *next_deadline* ms.
- *set_period(unsigned long period)* is a 'shortcut' routine that

can be used to avoid having to call `set_deadline()` repeatedly in cases where strictly periodic behaviour is required. The effect is identical to that of calling `set_deadline(period)` at the start of each period.

Unpoliced use of `set_sharetokens()` by applications could wreak havoc because any process could freely increase its number of sharetokens, resulting in ‘arms races’ and sharetoken inflation. Therefore, `set_sharetokens()` and `get_sharetokens()` are only available to the superuser. A user level API that builds on these calls but provides a level of policing is discussed in section 4.4.

On the other hand, `set_deadlinetype()`, `set_deadline()` and `set_period()` can be used directly by applications. Applications have little incentive to ‘cheat’ by setting over-early deadlines because deadlines are only used in the waking queue when processes have already blocked and missed some of their CPU allocation. Therefore, setting over-early deadlines would at best merely increase the rate at which a process recovers time it has already lost.

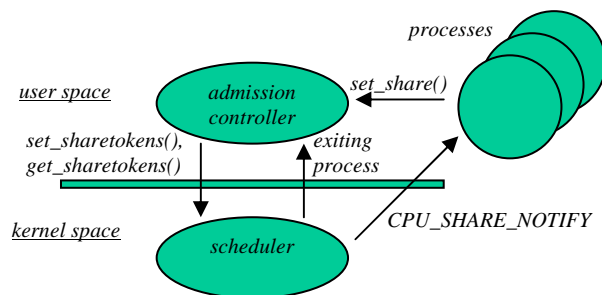
4.4 The Admission Control Module and its API

To prevent misuse by applications as discussed above, sharetoken allocation is managed and policed by a distinguished intermediary module called the *admission controller* (see figure 2). This is a user space process that runs with root privileges; other processes communicate with it via an inter-process communication mechanism².

The admission controller’s API comprises the following:

```
int set_share(int pid, int desired_share, int minimum_share);
int get_share(int pid);
```

The `set_share()` and `get_share()` functions are exported by a stub library linked into the client process and are translated by the library into IPC calls on the admission controller. The client process uses `set_share()` to specify a desired share of the CPU for the specified process as a percentage of the total CPU bandwidth (i.e. *not* directly in terms of sharetokens), and also a minimum acceptable share. The function returns the actual share allocated, or 0 if the request cannot be accommodated. If the function returns a value greater than 0, the client process is said to have ‘registered’ with the admission controller. Clients can call `get_share()` to discover the CPU share currently allocated to a specified process.



² Note that as the admission controller runs as a user space process, the admission control policy is decoupled from the scheduler itself and therefore straightforward to change.

Figure 2: The admission control architecture.

For these calls to succeed, the caller of `set_share()` and `get_share()` must have standard UNIX permissions over the specified process `pid`—i.e. the caller must either be `pid` itself or be in `pid`’s process group. This latter possibility encourages a use-pattern in which per-application ‘supervisor’ processes ‘sub-allocate’ a per-application set of CPU shares among a set of per-application processes according to some application-specific policy. For example, a supervisor for a multi-window video surveillance application might sub-allocate a given per-application share among processes that display the individual video windows. In such an application an appropriate policy could be to dynamically sub-allocate the majority of the shares to the window in which the mouse pointer is currently situated.

When new resources become available (e.g. when another process downgrades its allocation or exits³), the controller, depending on its policy, can allocate more CPU shares to processes that are not currently receiving their desired share. Conversely, when resources become scarce, the controller can reduce the share of processes that are currently taking more than their `minimum_share` parameter (as a rule, the controller should try to ensure that no process will drop below its specified minimum share, but whether this is enforced is a matter for individual admission policies—see below).

Finally, processes are asynchronously notified of CPU share updates by means of the `CPU_SHARE_NOTIFY` signal, for which they can optionally provide a handler. This signal carries information on modified share allocations and thus gives applications the basic mechanism they need to adapt to varying processing resources.

4.5 Admission Control Policy

To facilitate the management of CPU shares, our currently implemented admission control policy divides the total CPU resource into N nominal shares, each represented by one sharetoken. Then, any process requiring a percentage s of the CPU is allocated $\lceil (s * N) / 100 \rceil$ sharetokens. As long as it will result in $\leq N$ sharetokens having been allocated, processes are allocated sufficient sharetokens to satisfy their `desired_share` parameter; if this cannot be achieved, sufficient sharetokens are allocated to satisfy the `minimum_share` parameter. Eventually, overload will be reached when the system has allocated more than N sharetokens, causing the CPU share of existing processes to decrease. For example, if the number of sharetokens allocated is $N + y$, a process that initially held a share of x / N , will now only receive a share of $x / (N + y)$.

A process registering when the system is already in overload is only allocated any sharetokens if the allocation required to satisfy its `minimum_share` parameter does not cause any existing process to drop below its `minimum_share` level. Assuming allocation is possible, a new process requiring a `minimum_share` of s will be allocated $\lceil (s * n) / (1 - s) \rceil$ sharetokens where n is the total

³ The admission controller employs a private user/ kernel interface, not discussed further in this paper, to learn from the OS about exiting processes.

number in current circulation⁴.

When a process holding x sharetokens terminates, there are three possibilities to consider:

- The total number of allocated sharetokens is $\leq N$ (i.e. underload). In this case no explicit action is taken; all processes are already running at their desired CPU share.
- The total number of allocated sharetokens is $>N$, but reduces to $\leq N$ when the exiting process's x sharetokens are subtracted. In this case the admission controller increase the share of some processes that are currently allocated less than their *desired_share* parameter; the available x sharetokens are allocated to such processes in order of their registration.
- The total number of allocated sharetokens is $>N$ and remains so after the exiting process's x sharetokens are subtracted (i.e. overload). In this case no explicit action is taken; all processes automatically obtain an increase in their share of CPU time.

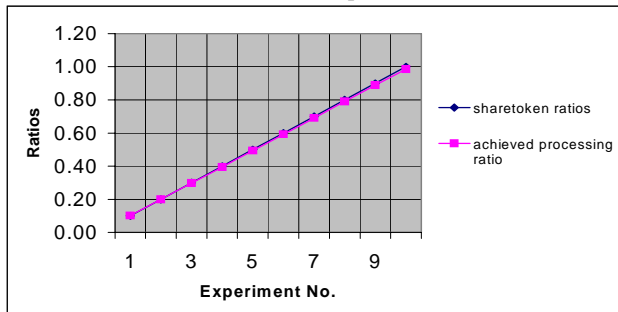
Clearly, this admission control policy makes somewhat arbitrary choices in certain areas—e.g. the initial selection of N , and the choice to allocate newly available sharetokens in order of registration. However, we have found it adequate in the majority of circumstances we have encountered to date.

5. EXPERIMENTAL EVALUATION

The following experiments were all executed on an Intel Pentium 2 PC rated at 350 MHz and with 64 Mb of RAM. The operating system used was Red Hat Linux version 7.2, kernel version 2.4.2.

5.1 Static Sharetoken Allocation

To verify that the algorithm yields CPU time in accordance with the number of allocated sharetokens, two processes P_1 and P_2 were run in parallel for 120 seconds, with varying sharetoken ratios (i.e. the number of sharetokens allocated to $P_1 \div$ the number of sharetokens allocated to P_2) in successive experiments. Each of P_1 and P_2 execute a loop continuously and the number of times the loop is executed is recorded for each process. However, while P_1 's loop is CPU bound, P_2 's contains a synchronous write of 4K of data to a disc file (this implies that the write occurs directly to the disc, not just to the buffer pool; it thus causes the process to block). P_2 performs the write every 5 to 7 seconds depending on the sharetoken ratio in the current experiment.



⁴ Derivation: if n sharetokens have already been allocated, then for the new process to obtain a share s of CPU time, we should have $x = s(n + x) \Rightarrow x(1 - s) = (s * n)$, where $\lceil x \rceil$ is the number of sharetokens to be allocated, from which the above follows.

Figure 3: Achieved processing ratio versus allocated sharetoken ratio.

As shown in figure 3, in spite of blocking, the CPU time obtained closely follows the allocated sharetoken ratio.

5.2 Dynamic Sharetoken Allocation

To verify that the scheduler is well-behaved under dynamic allocation of sharetokens, two processes P_1 and P_2 , each of which continuously execute a CPU bound loop, were allocated the same number of sharetokens and executed in parallel for 120 seconds. After this time, P_2 reduced its number of sharetokens to half for another 120 seconds and then returned to its initial allocation. The number of loops executed by each process was measured at 10 second intervals.

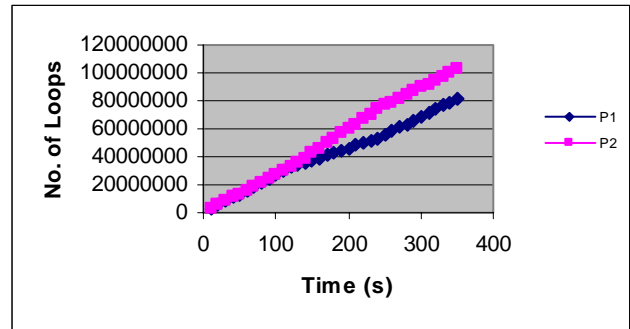


Figure 4: Variation of CPU time under dynamic allocation of sharetokens.

Figure 4 shows the progress of the processes in terms of number of loops executed over time. From the slopes of the different parts of the graph, it can be seen that when P_2 's sharetoken share was reduced to half of that of P_1 , the progress of P_2 reduces to half the rate. Later on when the initial shares are re-established, P_2 returns to the same progress rate as P_1 .

5.3 Impact of Deadline-driven Periodic Processes

An important goal of our scheduler is to honour the deadlines of deadline-driven periodic processes in underload situations, while not unduly penalising conventional processes on overload. To verify that this goal is achieved, three deadline-driven periodic processes D_1 , D_2 and D_3 were executed in the presence of a conventional process P_1 .

In this experiment, P_1 executes a loop continuously and uses as much CPU time as it can get. D_1 and D_2 and D_3 , all of which have periods of 200 ms, wake up at the beginning of each period, try to perform their processing and then block. If they cannot obtain their required processing time during any given period they are considered to have 'missed a frame'. D_1 and D_2 were made to process for 20% and 15% respectively of their periods (this was considered a suitable time to represent realistic media frame processing), while D_3 's processing time was varied between 2.5 % and 40% of its period on successive experimental runs. On each run, the total number of frames missed by all three deadline-driven processes was recorded as was the number of loops executed by P_1 . D_1 , D_2 and D_3 , were respectively allocated 4, 3 and 5 sharetokens while P_1 was allocated 8. These can be

considered to represent CPU proportions of 20%, 15%, 25% and 40% respectively⁵.

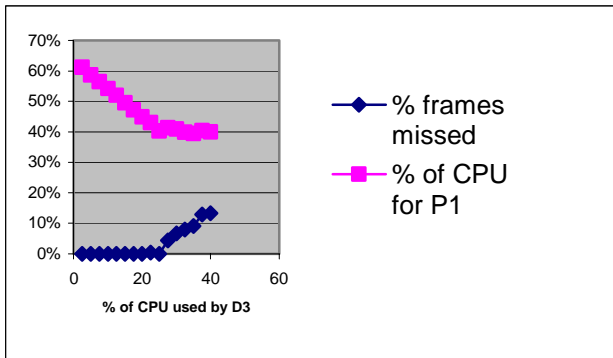


Figure 5: Impact of deadline-driven periodic processes.

Figure 5 shows the results. No frames were missed in underload situations, and the number of loops executed by P₁ decreased steadily as load was increased. As overload was reached, with D₃ using 25% of CPU time, a few frames were missed. With further increase in the CPU requirement of D₃, the number of loops for P₁ stabilised to near 40% but the number of missed frames increased progressively as D₃ claimed more and more CPU time. This confirms that the scheduler honours the deadlines of deadline-driven periodic processes in underload situations while not penalising conventional processes to the benefit of deadline-bound processes if the latter make excessive demands on the CPU.

5.4 Impact of Heterogeneous Application Mix

In this section we compare the performance of RRWS with the standard Linux scheduler when running a mix of batch mode and ‘real-world’ real-world continuous media applications. Batch mode processes are represented by a simple non-blocking program that uses as much CPU time as it can get, while continuous media processes are represented by an MPEG player running with a frame rate of 15 frames per second. In each experimental run, one instance of the MPEG player was executed together with a varying number (between 1 and 4) of instances of the batch program. All processes were executed at the same priority under the standard Linux scheduler (default user priority), and with the same number of sharetokens under RRWS. Each experiment lasted for 900 seconds and the total number of frames successfully displayed by the MPEG player was noted (see figure 6a) together with the total number of loops executed by all the batch applications (see figure 6b).

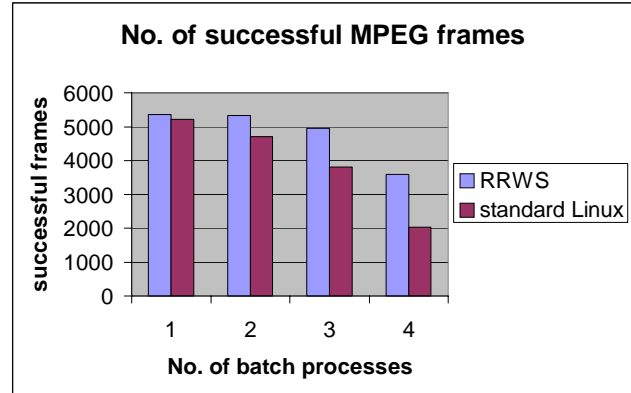


Figure 6a: Numbers of successfully played MPEG frames.

The results demonstrate that RRWS generally performs better than the standard Linux scheduler on both metrics. Interestingly, as the number of batch applications is increased, the difference in performance between RRWS and standard Linux in figure 6a seems to grow.

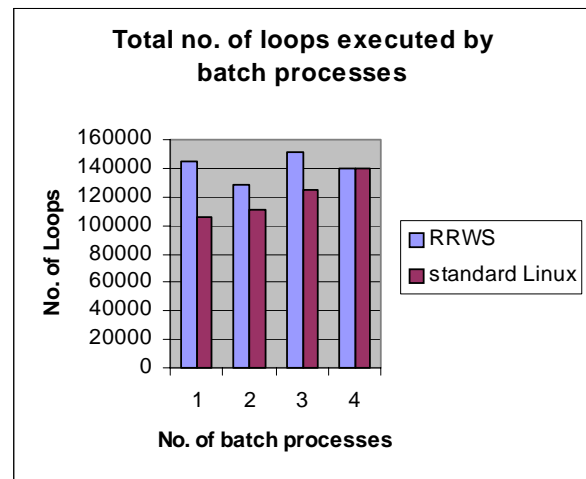


Figure 6b: Numbers of loops executed by batch processes.

5.5 Scheduling Overhead

In this final experiment we compared the overhead of RRWS with that of the conventional Linux scheduler. To achieve this, six processes, P₁-P₆, were executed concurrently for 30 seconds. Each executed a continuous loop, and the total number of loops executed by all processes was noted. The relative scheduling overhead was evaluated simply by comparing the total number of loops executed under each scheduler.

In figure 7, experiment 1 shows the result when there is no blocking, experiment 2 shows the result when 3 of the processes perform blocking calls regularly (about one blocking call for every 50 ms of processing) and experiment 3 shows the results when experiment 2 is repeated in the presence of two deadline-driven processes. It can be seen that for the first experiment, with no blocking, our scheduler has lower overhead than the standard Linux scheduler.

⁵ In practice, however, the system also runs a number of system processes each with 1 sharetoken. As these execute occasionally, the available CPU time for our 4 processes will be a little less than expected.

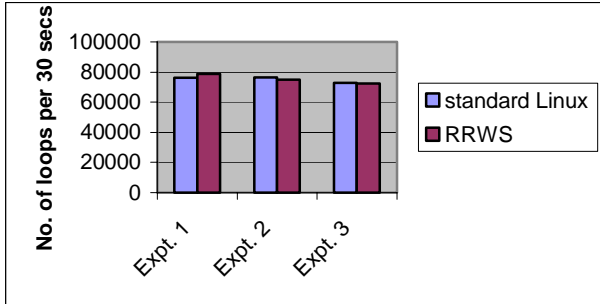


Figure 7: Comparative overhead of RRWS and the standard Linux scheduler.

Experiments 2 and 3 involve blocking calls, and thus bring the waking queue into play with its associated $O(n)$ addition and removal operations (at least for the deadline-driven processes). Here we see that, despite our unoptimised implementation⁶, our scheduler performs within 2% of the conventional Linux scheduler.

6. RELATED WORK

Proportional-share schemes using tickets [Waldspurger 94 & 94b] offer a straightforward approach to resource sharing among processes. Additionally, a process's resource share can easily be varied by changing its number of tickets. [Waldspurger 94] describes the use of *lottery scheduling* for allocating CPU bandwidth. Lottery scheduling is only statistically deterministic, and thus limited in its predictability. In contrast, *stride scheduling* [Waldspurger 94b] makes use of tickets in a deterministic way. RRWS is similar to stride scheduling in that *i)* it is deterministic, and *ii)* it yields a per-process scheduling share that is proportional to the number of tickets (sharetokens) held by each process. However, the overhead of RRWS should be significantly less than that of stride scheduling because, unlike the latter, RRWS does not need to perform an expensive recalculation of strides at each scheduling point. In addition, RRWS has explicit support for deadlines while stride scheduling does not. Furthermore, RRWS differs from stride scheduling in its handling of blocked processes. In stride scheduling, when a process wakes it only benefits from the amount of time left in its previous quantum whereas in RRWS a waking process can additionally recover whole quanta that it missed while it was blocked.

In another family of scheduling approaches, attempts have been made to subdivide applications into classes, with different scheduling policies used for each class [Goyal 96, Ford 96]. [Goyal 96] subdivides in terms of hierarchically partitioned classes and subclasses and allocates CPU times to nodes in the hierarchy using *start-time fair queuing*. Like RRWS, this allocates CPU time in such a way as to ensure proportional fairness. However, it does not consider deadlines. Therefore, when two blocked processes wake up they will receive the CPU according

⁶ It should be noted that we have not yet made any serious attempt to optimise our implementation. In particular, although we are not using the Linux run queue for scheduling, we have not interfered with the removal/addition of processes from/to the run queue, so as not to disturb other parts of the system. Therefore our current implementation incurs a significant amount of redundant context switch overhead.

the share that they have not received so far, without concern for which one has the closest deadline. [Ford 96] implements a number of virtual schedulers by having schedulers donate their CPU times to processes. However, this work only provides a general framework for equipping different classes of application with different scheduling schemes. As such, it does not provide any particular scheduling policy.

Nieh's SMART scheduler [Nieh 97] attempts to schedule both conventional and real-time processes through a common mechanism on the basis of virtual finishing time. However, it focuses on enabling real-time tasks to meet their deadlines and does not consider any form of readjustment according to changing availability of resources. In addition, it carries significant overhead; each scheduling point involves an $O(n^2)$ insertion sort involving all process descriptors, plus an EDF admission test per insertion.

A later offering from the same researchers, Virtual-time Round Robin [Nieh 01], is, like our RRWS scheme, based on the principle of weighted round robin. In terms of overhead, VTRR provides $O(1)$ scheduling whereas RRWS is $O(n)$ in the worse case. However, as discussed in section 2.4 we do not expect this to be significant in practice. In terms of facilities offered, there are three significant differences between VTRR and RRWS. First, VTRR does not explicitly support deadlines as does RRWS. Second, VTRR gives processes their proportional share of CPU time over longer time intervals, resulting in more jitter for periodic processes. For example, consider a run queue with a process P having 100 shares followed by two processes, Q and R, each with 4 shares. In VTRR, for the first 12 quanta, P, Q and R would receive an equal amount of CPU time; then, for the next 96 quanta Q and R will receive no CPU time at all. In RRWS, on the other hand, Q and R would never have to wait for the CPU for more than 26 quanta.

The third difference between VTRR and RRWS lies in the way they deal with blocked processes. In VTRR a waking process is placed back in the run queue and has to wait again for its turn. This may cause lower-share I/O bound processes to obtain less than their proportional share of CPU time. In RRWS, on the other hand, waking processes go to the waking queue where they recover (some of) their lost time. Furthermore, a process whose turn is reached while in a blocked state can accumulate up to one quantum to be used in the waking queue and further quanta to be used up gradually in the sharetoken queue.

Move-To-Rear List Scheduling as used in Eclipse [Bruno 98] is also related to our proposal. Like RRWS, MTR allows waking processes to recover unused time from their last quantum (during which they blocked) but does not support longer term recovery as does RRWS. Also, the notion of deadline is lacking in MTR. Furthermore, like VTRR, MTR allocates CPU time in bursts and may thus cause a high degree of jitter.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented *round robin scheduling with sharetokens*, a proportional-share scheduler that supports the co-existence of deadline-driven periodic processes and conventional processes. RRWS is efficient and deterministic, and it explicitly supports both deadlines and the adaptive capabilities of continuous media applications. Our experimental results to date

indicate that CPU bandwidth allocation is proportional to the number of sharetokens held even in the presence of blocking I/O, that the scheduler offers graceful degradation to conventional processes in case of overload, and that deadline-driven periodic processes are well supported in underload. Furthermore, as mentioned in section 3.2, we believe that an extension to RRWS (currently under investigation) should enable graceful degradation for deadline-driven processes under overload.

A planned extension is to add to our scheduler's API to better support portability across different CPU speeds. In particular, we would like applications to be able to specify their requirements in terms of statements like 'execute this code sequence in this (real) time window' rather than in terms of requesting a share of some *a priori* unknown CPU capacity. In the approach we are currently exploring, processes call a *begin()* routine at the start of a time critical section of computation which is likely to be similarly structured in the future (e.g. a period in the case of a periodic process), and an *end()* routine on its completion. The *end()* call specifies how long the computation should ideally have taken. Based on this information the admission controller updates relative shares to attempt to converge on a share assignment that will equitably balance the future needs of its managed processes.

A second area of planned development is the integration of our Linux-based RRWS implementation into the user-level scheduler framework reported in [Coulson 01]. This framework employs 'virtual processors', which we are implementing as RRWS-scheduled kernel threads, to support multiple user-level schedulers that can coexist in the same address space and simultaneously offer a selection of different scheduling policies to user-level threads. The framework is, in turn, integrated into a multimedia-capable distributed middleware platform [Coulson 02] that provides high-level support for networked multimedia applications. We expect RRWS to be a key factor in enabling this platform to provide predictable and adaptive quality of service to its applications.

REFERENCES

[Beck 98] Beck, M., Bohme, H., Dziaza, M., Kunitz, U., *Linux Kernel Internals*, Reading, MA, Addison-Wesley 2nd ed., 1998.

[Bruno 98] Bruno, J., Gabber, E., Özden, B., Silberschatz, A., *The Eclipse Operating System: Providing Quality of Service via Reservation Domains*, Proc. Usenix Annual Technical Conference, pp 235-246, 1998.

[Coulson 01] Coulson, G., Moonian, O., *A Quality of Service Driven Concurrency Framework for Object-Based Middleware*, Concurrency Practice and Experience (to appear), 2001.

[Coulson 02] Coulson, G., Baichoo, S., Moonian, O., *A Retrospective on the Design of the GOPI Middleware Platform*, to appear, ACM Multimedia Journal, 2002.

[Ford 96] Ford, B., Susanla, S., *CPU Inheritance Scheduling*, Proc. USENIX Association Symposium on Operating Systems Design and Implementation 2, pp 91-105, 1996.

[Goyal 96] Goyal, P., Guo, X., Vin, H.M., *A Hierarchical CPU Scheduler for Multimedia Applications*, USENIX Association Symposium on Operating Systems Design and Implementation 2, pp 107-121, 1996.

[Haldar 91] Haldar, S., Subramanian, D.K., *Fairness in Processor Scheduling in Time-Sharing Systems*, ACM Operating Systems Review, Vol 25, No 1, pp 4-18, 1991.

[Liu 73] Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, JACM, Vol 20, No 1, pp 46-61, Jan. 1973.

[Nieh 93] Nieh, J., Hanks, G., Northcutt, D., Wall, G.A., *SVR4 UNIX Scheduler Unacceptable for Multimedia Applications*, Proc. Fourth International Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, UK, pp 35-48, Nov. 93.

[Nieh 97] Nieh, J., Lam, M.S., *The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications*, Proc. 16th Symposium on Operating System Principles, Vol 31, No 5, ACM Press, New York, pp 184-197, Oct. 5-8, 1997.

[Nieh 01] Nieh, J., Vail, C., Zong, H., *Virtual-Time Round Robin: An O(1) Proportional Share scheduler*, Proc. 2001 Usenix Technical Conference, June 2001.

[Northcutt 91] Northcutt, J.D., Kuerner, E. M., *System Support for Time-Critical Applications*, Proc. Second International Workshop on Network and Operating System Support for Digital Audio and Video, Springer Verlag Lecture Notes in Computer Science, Vol 614, pp 242-254, Heidelberg, Germany, Nov. 1991.

[Silberschatz 98] Silberschatz, A., Galvin, P., *Operating Systems Concepts*, Addison Wesley, 1998.

[Steer 99] Steer, D.C., Goel, A., Gruenberg, J., McNamee D., Pu, C., Walpole, J., *A Feed-back Driven Allocator for Real-Rate Scheduling*, USENIX Association Symposium on Operating Systems Design and Implementation 3, pp 145-158, 1999.

[Waldspurger 94] Waldspurger, C.A., Weihl, W. E., *Lottery Scheduling: Flexible Proportional-Share Resource Management*, USENIX Association Symposium on Operating Systems Design and Implementation, 1994.

[Waldspurger 94b] Waldspurger, C.A., Weihl, W.E., *Stride Scheduling: Deterministic Proportional-Share Resource Management*, Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, Cambridge, MA, 1995.