

SHRED: a CPU Scheduler for Heterogeneous Applications

Oveeyen Moonian and Geoff Coulson

Distributed Multimedia Research Group

Lancaster University

Lancaster

+44 1524 593054

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

ABSTRACT

General purpose workstations must support a wide variety of application characteristics; but it is hard to find a single CPU scheduling scheme that satisfactorily schedules processes from all types of applications. It is particularly difficult to get periodic deadline-driven continuous media processes to satisfactorily co-exist with others. A number of schemes have been proposed to address this issue, but these all suffer from one or more of the following limitations: i) unacceptable inefficiency, ii) non-determinism (i.e. introducing significant burstiness or jitter), iii) inability to explicitly support deadlines (so that deadlines may be missed even when the CPU is underloaded). This paper presents “SHRED (SHaretokens, Round-robin, Earliest-deadline-first, Deferred-processing)” — an efficient, proportional-share, deterministic, scheduling scheme that enables periodic deadline-driven processes to meet their explicit deadlines wherever possible, and degrades gracefully and adaptively when this is not possible. The scheme simultaneously ensures that non-deadline processes always obtain their fair share of CPU time whether in conditions of underload or overload.

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 commodity 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 (which, just because they are not real-time are not necessarily less *important*), while the second (exemplified by UNIX SVR4) can additionally lead to unpredictable behaviour on overload, allowing runaway real-time activities to cause system services to lock up and the user to lose control of the machine [Nieh 93].

As a result, significant research (e.g. [Waldspurger 94 & 94b], [Goyal 96] [Ford 96], [Nieh 97 & 01], [Bruno 98], [Duda 99], [Banachowski 02]) has been carried out on new scheduling approaches that attempt to do better. Nevertheless, as discussed in section 5, all or most of these schemes suffer from one or more of the following deficiencies: they either incur high overhead, or are non-deterministic (and thus unable to adequately support real-time processes without unacceptable jitter), or do not explicitly support periodic deadline-driven processes (and thus do not

adequately support continuous media). The specific drawback of schemes that do not explicitly support deadlines is that there can be no guarantee that periodic processes will be able to meet their deadlines even in underload conditions.

In this paper we present a novel approach to supporting continuous media applications in a workstation environment that suffers from none of these deficiencies. SHRED (SHaretokens, Round-robin, Earliest-deadline-first, Deferred-processing) is a simple scheme that combines elements of Stride [Waldspurger 94b], Weighted Round-robin [Silberschatz 98], and Earliest Deadline First [Liu 73] scheduling to yield an efficient, proportional-share, deterministic, scheduling scheme that enables periodic deadline-driven processes to meet their explicit deadlines wherever possible, and degrades gracefully and adaptively when this is not possible (thus exploiting the naturally adaptive potential of continuous media applications [Northcutt 91]). The scheme simultaneously ensures that non-deadline processes always obtain their fair share of CPU time whether in conditions of underload or overload. Finally, the ‘deferred processing’ element of the scheme ensures that I/O bound processes that miss out on their fair share of CPU time because they voluntarily block are able to recover lost time later and thus maintain high levels of interactivity with a relatively low share of CPU time.

The remainder of the paper is structured as follows. Section 2 describes the scheduling algorithm, and section 3 discusses our implementation in Linux, including a policy-driven admission control architecture. Section 3 also discusses our approach to continuous media adaptation (i.e. policing applications and notifying them of altered CPU shares so that they can adapt). Then section 4 offers an experimental evaluation of our implemented scheduler, and section 5 compares our design with related research. Finally, we present concluding remarks in section 6. Appendix A analyses the way in which the scheduler treats periodic deadline-driven processes. It demonstrates that, under SHRED, these processes never miss their deadlines except under conditions of overload.

2. SHRED

2.1 The Basic Algorithm

SHRED is presented incrementally in this and the following subsections (2.2, 2.3 and 2.4). Initially, in this section, we make the (unrealistic) assumption that processes never block, and then relax that assumption in the subsequent subsections.

On process creation, each process is allocated a number of so-called *sharetokens* to represent its required proportion of CPU time. These are analogous to tickets in lottery scheduling [Waldspurger 94]. The proportion of CPU time allocated to a

process P is the ratio of P 's sharetoken count to the total count of sharetokens currently held in the system (i.e. by P and by all other processes). Note that the number of sharetokens in the system is not static: it varies as processes come and go.

All sharetokens are held in a circular *sharetoken queue* managed by the scheduler. The scheduler works in simple round-robin fashion: it selects the 'current' sharetoken for the next quantum of CPU time; and then moves the 'current' pointer to the next sharetoken in preparation for the next time the scheduler is invoked. The n sharetokens held by each process are distributed in as evenly spaced a manner as possible throughout the sharetoken queue to minimise the burstiness or jitter experienced by any process. Figure 1 shows the $n=5$ sharetokens owned by a newly created process P_3 being added in such a manner to an existing queue configuration.

Even spacing is achieved as follows: assuming there are already x sharetokens in the queue when the sharetokens of a new process is added, we set a variable 'spacing' to $(x+n) / n$, and then place the sharetokens at multiples of *spacing* rounded to the nearest integer. For example, if (as in figure 1) a process with 5 sharetokens is added to a queue with 8 sharetokens, *spacing* will be $(8+5) / 5 = 2.6$, and the 5 new sharetokens will be placed at positions $\text{round}(2.6)$, $\text{round}(2 * 2.6)$, $\text{round}(3 * 2.6)$, $\text{round}(4 * 2.6)$, and $\text{round}(5 * 2.6)$; i.e. at positions 3, 5, 8, 10, and 13.

Additionally, the first of the n new sharetokens is placed at some random point in the queue (e.g. as determined by the round-robin scheduler's current queue pointer position). This helps ensure that the sharetokens from each process automatically remain (on the whole) evenly spaced as the queue grows and shrinks as processes are created and deleted.

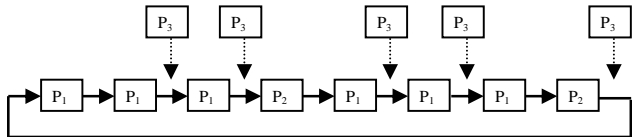


Figure 1: An example sharetoken queue.

Given this arrangement, after each round-robin cycle through the queue each sharetoken will have received one quantum on behalf of its owning process. Hence a process owning n sharetokens will obtain the CPU for n quanta per cycle. In figure 1, for example, processes P_1 , P_2 and P_3 hold 6, 2 and 5 sharetokens respectively, and therefore in each cycle respectively obtain that number of quanta.

2.2 Dealing with Blocking Processes

The sharetoken queue discussed above is *static* in nature: it only changes when processes are created or deleted. To enable the scheduler to view only *runnable* sharetokens (i.e. sharetokens whose owing process is currently runnable), sharetokens additionally participate¹ in a dynamic circular *run queue* that only holds sharetokens that are currently runnable. It is on this run queue that the round-robin scheduler actually operates.

The algorithm to *block* a process is straightforward: it simply

iterates through the process's sharetokens and, for each one, unlinks it from the run queue while leaving it in the static queue.

The algorithm to *unblock* a process is more complex. The requirement here is to link each of the unblocking process's sharetokens, *st*, into the run queue such that *st*'s neighbours are those sharetokens already in the run queue that are closest to *st* in the static queue. To achieve this, every sharetoken in the static queue (i.e. every sharetoken in the system) has an integer *index* field which is set according to its position in the static queue (the sharetoken at the front of the queue has the lowest value)². Given these indices, to unblock a process we simply *merge* the process's sharetokens into the run queue in such a way that an absolute ordering of the sharetokens in the run queue is maintained.

Appendix B presents the algorithms for blocking and unblocking in detail. These are embodied in the *block()* and *unblock()* routines that are referred to in the rest of the paper. In addition, we discuss the complexity of these algorithms in section 2.5.

2.3 Supporting Deadline-Driven Processes

To support deadline-driven processes we introduce a third queue into the scheme, called the *waking queue*, after the fashion of 'Virtual Round Robin'[Haldar 91].

This works as follows: whenever a deadline-driven process wakes up that had voluntarily blocked before the end of its previous quantum, an additional, 'temporary', sharetoken is allocated for it on the waking queue. The quantum length associated with this is the standard quantum length minus what the process used of the previous quantum in which it blocked. The sharetoken is inserted into the queue in *earliest deadline first* (EDF) order.

Having introduced the waking queue, the basic round-robin scheduling algorithm is modified to *always look first in the waking queue for the 'current' sharetoken before proceeding to examine the run queue*. 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 queue. When the process has exhausted this sharetoken's quantum the sharetoken is deleted from the waking queue. As we demonstrate in Appendix A, this arrangement guarantees that periodic deadline-driven processes always meet their deadlines in cases of underload (for the definition of 'underload' and 'overload' see section 3.5). But, an equally important property is that the waking queue arrangement can never starve sharetokens in the run queue. This is because the waking queue sharetokens are temporary so that when their quantum has expired the corresponding process must subsequently be scheduled from the run queue in the normal way.

Another important benefit of the scheme is that *unblock()*—which is of linear complexity (see section 2.5)—need not be invoked when a process unblocks having previously voluntarily blocked while executing a sharetoken from the waking queue. This is a corollary of the fact that the waking queue is always examined first by the scheduler: an unblocking process would never, under these circumstances, be scheduled from the run queue and therefore it would be pointless to replace its sharetokens there by

¹ To enable this, sharetokens have separate pairs of 'previous' and 'next' pointers relating to each of the queues.

² Note that it is necessary to update these indices every time a process is created or destroyed—i.e. added to or removed from the static queue. The overhead of this, however, is negligible as it is necessary anyway to scan the static queue on process creation/ destruction.

calling *unblock()*. Furthermore, for similar reasons, *block()* and *unblock()* do not need to be called on subsequent blockings and unblockings during that same quantum, either.

2.4 Deferred Processing: Fairness for Voluntarily Blocking Processes

2.4.1 Motivation

The basic round-robin algorithm as so far described ensures strict proportional share over whole numbers of round-robin cycles, *assuming all processes are always runnable*. However, when processes block, their current quantum may not have been fully consumed and, furthermore, they may miss one or more whole sharetoken-turns while they are blocked. This is at odds with the traditional workstation scheduling philosophy of favouring voluntarily blocking, I/O bound, processes over CPU bound processes—the idea being to maintain high responsiveness for the former. To restore the balance in favour of I/O bound processes, we introduce two additional mechanisms into the scheme, under the heading of ‘deferred processing’. These are described in the following two subsections.

2.4.2: Part-Quantum Deferred Processing: the Waking Queue

We have already discussed the waking queue in the context of supporting periodic deadline-driven processes. For deferred processing of non-deadline-driven processes it works in an exactly similar way except that the temporary sharetoken (which, as for the deadline-driven case, is issued when a process voluntarily blocks before the end of a standard quantum) is placed at the *rear* of the waking queue. This ensures that I/O bound processes that miss part of their quantum are appropriately compensated, but not at the expense of causing periodic deadline-driven processes to miss their deadlines (see Appendix A for proof of this). It also means that non-deadline-driven processes can avoid calling *unblock()* and *block()* when unblocking/blocking from the waking queue, as described in section 2.3.

2.4.3 Multi-Quantum Deferred Processing: the Accumulated Time Mechanism

The above-described waking queue scheme adequately addresses cases in which a waking process has lost time from its *current* sharetoken-turn through voluntary blocking. The *accumulated time* mechanism opens up the further possibility of deferred processing in cases where the process additionally misses one or more whole sharetoken-turns while it is blocked. The scheme is as follows:

- we define a notion of ‘accumulated time’ as the standard quantum length multiplied by the number of times the waking process has had a sharetoken skipped³;
- when a process misses sharetoken-turns, we add some

³ We count skipped sharetoken-turns as follows: we maintain a counter that is incremented at the end of every cycle through the run queue. Then, every time a process blocks, we record the current value of this counter in the process’s descriptor. Whenever a process unblocks we can determine (approximately) how many cycles (and thus how many sharetoken-turns) the process has lost by comparing the recorded value with the current counter value. The scheme is approximate because it only counts full cycles—if the process unblocks part way through a cycle, the sharetoken-turns missed since the end of the previous cycle will not be counted. The scheme may undercount but not overcount.

fraction of its accumulated time to each of its subsequent sharetoken-turns until it has ‘caught up’.

To make this scheme workable, it needs to be configured with heuristically-determined *accumulation bound* and *quantum bound* parameters. ‘Accumulation bound’ refers to the maximum amount of accumulated time a process may accumulate. Although an accumulating process never takes 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 according to the scheme described above. Large quantum bounds will have a relatively larger impact but over a shorter period (and vice versa).

2.5 Complexity Analysis

Scheduling from the run queue involves selecting the current sharetoken and moving a pointer to the next sharetoken in the run queue. Similarly, scheduling from the waking queue involves selecting the process at the front of this queue. Both of these operations are $O(1)$. Therefore scheduling is carried out in all cases in $O(1)$ time.

Turning to the overhead of blocking and unblocking processes, as mentioned in section 2.2, *unblock()* and *block()* are not invoked when processes voluntarily unblock from/ reblock to the waking queue. Therefore, under these conditions, the overhead of blocking and unblocking is $O(1)$. When, however, processes block or unblock while executing a sharetoken from the run queue, the overhead of *block()* and *unblock()* is incurred. It can be seen by inspection (see Appendix B) that the complexity of *block()* is $O(n)$, where n is the number of sharetokens owned by the blocking process. By default, processes on creation are issued with one sharetoken, so these processes block in constant time; it is only processes that ask for more sharetokens that incur a proportionately higher overhead.

The basic overhead of *unblock()* is that of merging the unblocking process’s sharetokens into the run queue such that the indices of the sharetokens in the run queue are maintained in monotonically increasing order. Thus, following the standard complexity of such a merge, the complexity of unblocking is $O(m+n)$ where m is the number of sharetokens in the run queue, and n is the number of sharetokens owned by the unblocking process.

If an unblocking process needs to have a temporary sharetoken allocated (see section 2.4.2) for the waking queue, there is additional overhead associated with unblocking, the degree of which depends on whether the process is non-deadline-driven or deadline-driven. Adding a non-deadline-driven process to the waking queue is accomplished in $O(1)$ time as the sharetoken is simply placed at the rear of the queue. However, adding a deadline-driven process is $O(w)$ (where w is the number of sharetokens in the waking queue)—because the sharetoken must be inserted in the queue in deadline order. It is worthwhile emphasising that this overhead is *only incurred for deadline-driven processes*, which is arguably a reasonable price to pay for the value-added service of EDF scheduling.

Finally, we turn to the overhead of creating and deleting

processes. When *creating* a process, adding its sharetokens involves scanning the whole of the static queue (the indices also have to be reassigned during this scan) and is therefore an $O(n+m)$ operation (where n is the number of sharetokens owned by the new process, and m is the number of sharetokens already in the static queue). *Deleting* a process is $O(n)$ (where n is the number of sharetokens owned by the process being deleted). Of course, these operations are very rare in comparison to scheduling decisions and the blocking/ unblocking of processes; in fact the overhead that SHRED imposes on process creation and deletion is entirely negligible in most practical situations.

The above complexity analysis is summarised in the table below.

<i>Operation</i>	<i>Complexity</i>
scheduling a process	$O(1)$
blocking a process P	$O(1)$ if the process blocks while executing under a sharetoken from the waking queue; otherwise: $O(n)$ where n is the number of sharetokens owned by P
unblocking a non-deadline-driven process P	$O(1)$ if the process unblocks while executing under a sharetoken from the waking queue; otherwise: $O(n+m)$ where n is the number of sharetokens owned by P, and m is the number of sharetokens in the run queue:
unblocking a deadline-driven process	$O(1)$ if the process unblocks while executing under a sharetoken from the waking queue; otherwise: $O(u+w)$ where u is the cost of unblocking a non-deadline-driven process (as above), and w is the number of sharetokens in the waking queue
creating a process P	$O(n+m)$ where n is the number of sharetokens owned by P, and m is the number of sharetokens already in the static queue
deleting a process P	$O(n)$ where n is the number of sharetokens owned by P.

3. IMPLEMENTATION

3.1 Kernel Data Structures and Functions

Our prototype scheduler has been implemented under Red Hat Linux version 7.2, kernel version 2.4.2. The implementation closely follows the descriptions in section 2 and Appendix B. In addition to the data structuring machinery detailed in Appendix B, 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* records the next deadline (only used if *is_deadline_type* is true).

When the system boots, the static sharetoken queue is initialised with a single sharetoken for *init_task*. Subsequently, each newly created process is allocated one sharetoken by default (more can be added later as described below). 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 to add a sharetoken to the static queue.

3.2 Scheduler Implementation

The implementation of the scheduler itself closely follows the abstract description in section 2. Our implementation completely replaces the original Linux scheduler so that all system daemons as well as application processes are scheduled using SHRED (system daemons use the default allocation of a single sharetoken).

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

3.3 System Call API

The following new system calls enable user-level 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, it returns 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.

Of course, unpoliced use of `set_sharetokens()` by applications could wreak havoc because any process could freely increase its number of sharetokens, resulting in sharetoken inflation and ‘arms races’. Therefore, `set_sharetokens()` is only available to the superuser. The `get_sharetokens()` routine is also restricted in that it will only work for non-superusers if the caller has standard UNIX permissions over the specified `pid`—i.e. the caller must either be `pid` itself or be in `pid`’s process group (or `pid` can alternatively be 0). A suitably policed user-level API that builds on these calls is discussed in section 3.4.

On the other hand, the other three calls 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.

3.4 The Admission Control Module and its API

To prevent misuse by applications as discussed above, sharetoken allocation is managed and policed by an 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 mechanism. Note that as the admission controller runs as a user space process, the admission control *policy* is decoupled from the scheduler itself and is therefore straightforward to change. Our current admission control policy is described in section 3.5.

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. Clients use `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. Clients can call `get_share()` to discover the CPU share currently allocated to a specified process.

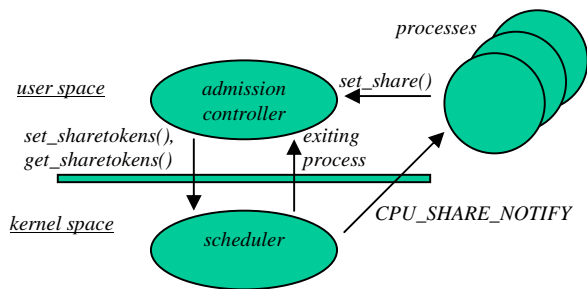


Figure 2: The admission control architecture.

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 a per-application ‘supervisor’ process ‘sub-allocates’ 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 the processes that display individual video windows. In such an application an appropriate policy might 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 exits⁴), 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, the API supports two new UNIX signal types to inform applications of resourcing changes that potentially affect them. The first of these, `CPU_SHARE_NOTIFY`, asynchronously notifies processes (that registered a handler) of CPU share updates, and carries information on modified share allocations, thus giving applications the basic mechanism they need to adapt to varying processing resources. The second, `CPU_OVERLOAD_NOTIFY` is issued whenever the CPU becomes overloaded (see next section for characterisation of ‘overload’). This is especially useful to periodic deadline-driven processes as it informs them of ensuing conditions under which deadlines may be missed.

3.5 Admission Control Policy

In this section we describe an example admission policy that attempts to prevent CPU overload while being flexible and fair to its clients. It is also, of course, perfectly viable to employ a null admission policy and rely on SHRED’s graceful degradation properties to encourage applications to police themselves. This is especially true in the single-user environments at which SHRED

⁴ The admission controller employs a private user/ kernel interface, not discussed further in this paper, to learn from the OS about exiting processes.

is primarily targeted.

Our 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. More precisely, if the number of sharetokens allocated is $N + m$, a process that initially held a share of n / N , will now only receive a share of $n / (N + m)$.

A process that calls *get_share()* when the system is already in overload is, according to our current policy, not allocated any additional sharetokens unless 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 is allocated $\lceil (s * x) / (1 - s) \rceil$ sharetokens where x is the total number in current circulation⁵.

When a process holding n sharetokens is about to terminate, there are three possibilities to consider:

1. The total number of allocated sharetokens is currently $\leq N$ (i.e. the system is in underload).
2. The total number of allocated sharetokens is currently $>N$, but will reduce to $\leq N$ when the exiting process's n sharetokens are subtracted.
3. The total number of allocated sharetokens is currently $>N$ and will remain so after the exiting process's n sharetokens are subtracted (i.e. the system is in overload).

Our policy here is as follows: In case 1 (underload) no explicit action is taken; all processes are already running at their desired CPU share. In case 2 the admission controller allocates the newly available shares among processes that currently have less than their *desired_share* parameter. These processes are ranked according to the time of their first call to *get_share()*, and each is topped up to its *desired_share* level until all the available shares are allocated. Finally, in case 3 (overload) no explicit action is taken; all processes automatically obtain an increase in their share of CPU time.

4. EXPERIMENTAL EVALUATION

4.1 Context

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 was Red Hat Linux version 7.2, kernel version 2.4.2. In all cases the quantum size used is identical to standard Linux quantum size of 50 ms.

⁵ 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.

4.2 Static Sharetoken Allocation

The purpose of this first experiment was to verify that SHRED allocates CPU time in proportion to the number of sharetokens allocated, regardless of whether processes block or not. Two processes P_1 and P_2 were executed in parallel for 120 seconds; varying sharetoken ratios (i.e. the number of sharetokens allocated to $P_1 \div$ the number allocated to P_2) were used in successive experiments. P_1 and P_2 both repeatedly execute a loop and the number of times the loop is executed is recorded for each process. While P_1 's loop is CPU bound, P_2 's contains a synchronous write of 4K of data to a disc file (this size makes the write occurs directly to the disc, not just to the buffer pool, and 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.

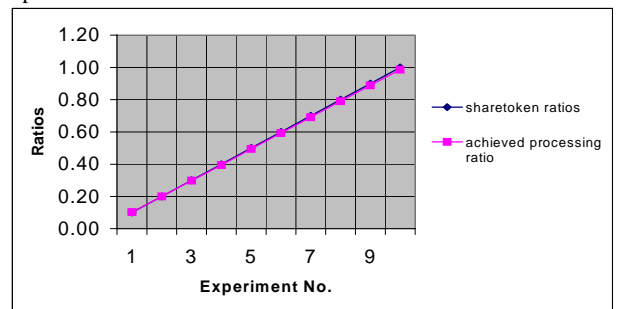


Figure 3: Achieved processing ratio versus allocated sharetoken ratio.

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

4.3 Dynamic Sharetoken Allocation

To verify that the scheduler is well-behaved under dynamic reallocation of sharetokens, two CPU-bound processes were executed in parallel and their relative sharetoken ratios were varied. The expectation was that the CPU time allocated to the two processes would vary accordingly. In more detail, the two processes, P_1 and P_2 , each of which continuously executes a tight CPU-bound loop, are initially given the same number of sharetokens. Then, after 120 seconds of execution, P_2 reduces its number of sharetokens to half for another 120 seconds and then returns to its initial allocation. The number of loops executed by each process are measured at 10 second intervals.

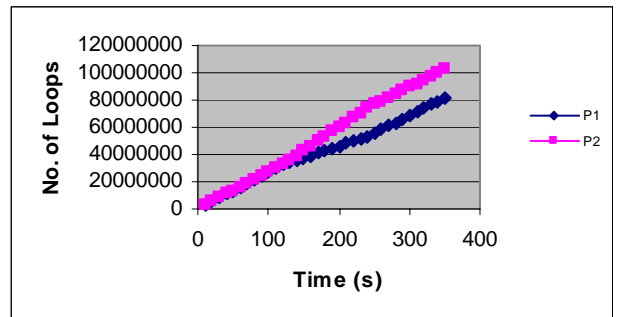


Figure 4: Variation of CPU time under dynamic allocation of sharetokens.

Figure 4 shows the progress of the two processes in terms of the 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 (at 120 secs), the progress of P_2 reduces to about half the rate. Later on when the initial shares are re-established (at 240 secs), P_2 returns to the same progress rate as P_1 . Thus we can see that the processes behave as expected according to their sharetoken allocations.

4.4 Impact of Periodic Deadline-driven Processes

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

In the experiment, P_1 continuously executes a tight loop and uses as much CPU time as it can get. D_1 , 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 per-period 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 fixed percentages of their period (20% and 15% respectively which was considered to represent a 'realistic' media processing scenario) while D_3 's processing time was varied over 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 .

The total CPU resource was represented by 20 sharetokens⁶ and D_1 , D_2 and P_1 , were respectively issued with 4, 3, and 8 sharetokens. This means that full utilisation of the CPU occurs when D_3 has 5 sharetokens (i.e., $4 + 3 + 8 + 5 = 20$). At this point the respective CPU proportions of D_1 , D_2 , P_1 , and D_3 are 20%, 15%, 40% and 25% (that is, D_1 , D_2 , and D_3 respectively execute for 20%, 15% and 25% of their periods). In the experiments, D_3 's sharetoken allocation was varied on successive experimental runs so as to represent a per-period execution time of between 2.5% (comfortable underload) and 40% (severe overload).

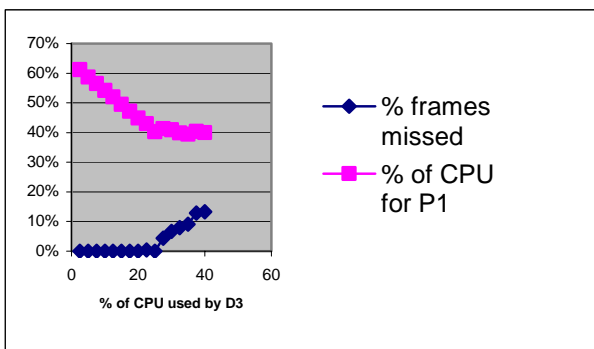


Figure 5: Impact of periodic deadline-driven processes.

⁶ In practice, however, the system also runs a number of system processes and daemons, each with 1 sharetoken. As these execute occasionally, the available CPU time for our 4 processes will be a little less than expected.

Figure 5 shows the results. No frames were missed in underload situations (i.e. where D_3 's CPU share was $< 25\%$), and the number of loops executed by P_1 steadily decreased as load was increased. As full utilisation was reached, with D_3 at 25%, frames began to be missed. With further increase in D_3 's CPU requirement, the number of loops executed by P_1 stabilised to near 40% but the number of missed frames increased progressively as D_3 claimed more and more CPU time. Thus we confirm that the scheduler honours the deadlines of periodic deadline-driven processes in underload situations while not unfairly penalising conventional processes to the benefit of deadline-bound processes if the latter make excessive demands on the CPU.

4.5 Impact of Heterogeneous Application Mix

In this experiment, we compared the performance of SHRED with that of the standard Linux scheduler while running a mix of batch mode applications and 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 execute 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. Under the standard Linux scheduler, all processes were executed at the same priority (default user priority). Under SHRED, all processes were similarly executed with the same number of sharetokens. Each experiment lasted for 900 seconds and the total number of frames successfully displayed by the MPEG player was recorded (see figure 6a), together with the total number of loops executed by all the batch applications (see figure 6b).

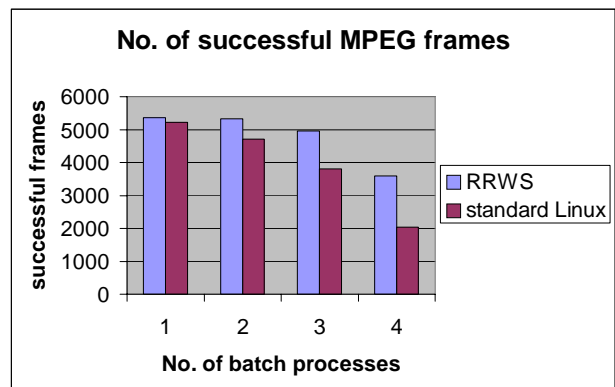


Figure 6a: Numbers of successfully played MPEG frames.

The results demonstrate that SHRED 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 SHRED and standard Linux in figure 6a apparently grows indicating superior scaling of SHRED.

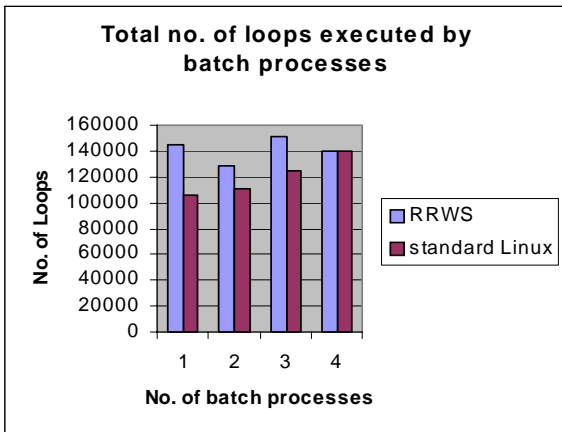


Figure 6b: Numbers of loops executed by batch processes.

4.6 Effect on Jitter

An important goal of multimedia systems is to minimise jitter, i.e. the difference in delay between successive output units (e.g. frames for a video player). Our scheduler has been evaluated for jitter control against the standard Linux scheduler as well as against two other proportional schedulers, namely the Stride scheduler and the VTRR scheduler. To avoid interference from the I/O subsystem and memory management involved with actual multimedia players, we have simulated real-time by having a process that wakes up every 400 ms, processes for some time then blocks waiting for its next period. In a first experiment we ran 16 such processes, with each running for 20.5 ms in each period, i.e. using about 5.1% CPU time, with a total of 82% being used by all 16 processes together. The mean jitter was recorded. The experiment was then repeated in the presence of a batch process that uses as much CPU time as it can get. Again the mean jitter was recorded. The results of both experiments are shown in figure 7.

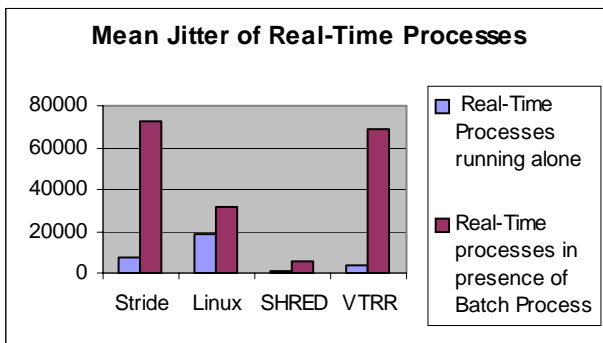


Figure 7 – Comparisons for mean Jitter

The results show that SHRED has lowest mean jitter in both cases, performing much better than all the three other schedulers. It is also noted that while Stride and VTRR performs reasonably well in the absence of the batch process, doing quite better than Linux, their jitter degrades considerably with the presence of the batch process. Their good performance in the first experiment may be due to the fact that they are dealing with all processes following the same pattern, but the presence of the batch process

interrupts the pattern. The number of loops executed by the batch process in the second experiment is shown in table 1 below.

Table 1

Scheduler	No. of loops for Batch Process
Stride	16970
Linux	17623
SHRED	17856
VTRR	31484

The results show that under Linux, Stride and SHRED, the amount of CPU time obtained by the batch process is similar, whereas with VTRR the batch process obtains more CPU time. This is also expected since VTRR uses virtual time, which ensures proportional share among active processes but does not compensate processes for the time they lose while blocked, whereas all the other three provide some means of compensating.

4.7 Scheduling Overhead

We have noticed that, interestingly, a lot of the works on scheduling do not measure scheduling overhead. Some works like [Nieh 01] perform overhead measurement at a micro-benchmark level, i.e. by using timestamps to measure the time taken to make a scheduling decision. We feel that this technique of measurement does not really reflect the full picture from a performance point of view, vis-à-vis applications. The actual performance of the scheduler in a working environment doesn't only depend on the time to make a scheduling decision, but also on the time for deleting and inserting processes in the runqueue. [Nieh 01] describes results of VTRR, wherein the average time at micro levels for a scheduling decision is about 100 times better than that of the standard Linux scheduler when 200 processes are running. This is very much to be expected since VTRR does not have to search the queue, as Linux does, at every scheduling decision. However when a process wakes up Linux inserts it in the queue in O(1) operations whereas VTRR needs O(n) operations. With 200 processes in the runqueue, every time a system process wakes up (which occurs very frequently for processes like the swapper, for example) VTRR has to perform 200 comparisons to insert it in the queue.

Our approach has thus been to measure the overhead as observed by the applications. In this final experiment we compared the overhead of SHRED with that of the three other schedulers, namely Stride, VTRR and the Linux native scheduler. A number of processes, each executing a loop continuously, are executed concurrently for 900 seconds. The start time and end time of each process is logged as well as the total number of loops executed by each process. From this data the average number of loops executed per second is calculated to yield a comparison of the relative scheduling overheads of the schedulers. The results are shown in table 2, for numbers of processes varying between 50 and 200 (in steps of 50).

Table 2

No of Processes	Stride	Linux	VTRR	SHRED
50	2397	2423.3	2427.2	2427.5

100	2392	2420.9	2426.9	2427.1
150	2384	2419.5	2426.5	2426.4
200	2355	2418	2426	2426

The results show that SHRED and VTRR have very similar scheduling overhead, which is marginally better than the Linux native scheduler. Stride scheduling has about 1.5% to 2.5% worse overhead than the three other schedulers. It is to be noted that the relative figures we obtained between the Linux native scheduler and Stride scheduler are consistent with the results listed in [Waldspurger 94b].

5. RELATED WORK

Proportional-share schemes using tickets [Waldspurger 94 & 94b] offer a straightforward approach to resource sharing among processes. *Lottery scheduling* [Waldspurger 94], in which processes hold varying numbers of lottery tickets and the process to be executed is the winner of the lottery, is only statistically deterministic, and thus limited in its predictability. In contrast, *Stride scheduling* [Waldspurger 94b] makes use of tickets in a deterministic way to improve predictability (as does SHRED). Following Waldspurger's works, other proportional schedulers [Nieh 97 & 01, Bruno 98, Duda 99] have been developed to support proportional CPU sharing. However, among these only Nieh's SMART scheduler [Nieh 97] has explicit support for deadlines. In all the others when two blocked processes wake up they will receive CPU time based on how far they are lagging on their fair share, without concern for which one has the closest deadline. SHRED differs from these schemes (except SMART) in that it provides explicit support for deadlines.

In another family of scheduling approaches, applications are subdivided 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 SHRED, this allocates CPU time in such a way as to ensure proportional fairness. However, it again does not consider deadlines. [Ford 96] implements a number of *virtual schedulers*. However, this work only provides a general framework for equipping different application classes 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, unlike SHRED, does not consider any form of readjustment according to changing availability of resources. Thus, under overload high priority processes are executed, whereas low-priority ones are simply dropped out. Having a proportional-share approach SHRED offers direct dynamic adaptation. Under overload, all processes are automatically allocated a reduced share, proportionately to the share they already hold. In addition, SMART 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 SHRED, based on the principle of

weighted round-robin. In terms of overhead, VTRR provides $O(1)$ scheduling as does SHRED. In terms of facilities offered, there are *three* significant differences between VTRR and SHRED. First, like the other schemes reviewed above, VTRR does not explicitly support deadlines as does SHRED. Second, because VTRR gives processes their proportional share of CPU time over longer time intervals, it is inherently bound to introduce 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 receive an equal amount of CPU time; then, for the next 96 quanta Q and R receive no CPU time at all. In SHRED, on the other hand, Q and R would never have to wait for more than 26 quanta. It is to be noted however, that the jitter we measured in section 4.6 is for each process holding only one sharetoken. The evaluation of the effect of uniform distribution over the sharetoken queue is being kept for future works. The third difference between VTRR and SHRED 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 causes lower-share I/O bound processes to obtain less than their proportional share of CPU time. In SHRED, on the other hand, I/O bound processes can recover their lost share through the waking queue and accumulated time mechanisms.

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

BVT [Duda 99] supports real-time by allowing processes to borrow (in advance) an amount of CPU time based on a *warp value*. This warp value is arbitrary and has to be decided by the user or application programmer. Therefore, when there are a number of real-time processes the user or application programmer has to use his judgment to decide on the warp value, whereas SHRED supports an actual deadline, which occurs as a natural feature of real-time processes. Besides, warp values are static throughout the lifetime of the process whereas deadlines are dynamic and are reset regularly.

The Best-effort scheduler Enhanced for Soft real-time Time-sharing (abbreviated to BEST) [Banachowski 02] makes use of features of Linux scheduler, but monitors process behaviour to identify real-time ones and assigns priorities to these based on their expected deadlines (which are themselves based on the monitored behaviours). Although BEST is useful when working on a PC having at most a few real-time processes, it cannot discriminate between processes having the same period. Besides, it can only support as many different priorities as the number existing in the Linux RT class. Thus it does not seem to provide an adequate approach for a workstation or larger system having a large number of real-time processes. Besides, since it assumes that real-time processes are purely periodic, it cannot cater for weakly periodic processes as defined by [Steinmetz 02]. Also, like all best-effort schedulers, the BEST scheme can allow the CPU to be oversubscribed, causing real-time processes to miss their deadlines, and non-real time processes to starve. Besides, like other best-effort schedulers it does not provide any

mechanism to allow real-time processes to adjust to different QoS-levels, thus it is not making use of the important adaptive characteristic of such applications.

6. CONCLUSIONS AND FUTURE WORK

This paper has presented SHRED, a deterministic, $O(1)$ proportional-share scheduler that supports the co-existence of periodic deadline-driven periodic processes and conventional processes. We have also described the system call API to the scheduler, and a user-level architecture that controls access to critical scheduler functions and supports ‘pluggable’ admission control policies. Our experimental results 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 periodic deadline-driven processes are well supported in underload. Besides we show that our scheduler ensures low jitter for real-time processes in a near-full load situation even in the presence of non-realtime processes.

In summary, the different mechanisms involved in SHRED, and their various roles, are as follows. First, the combination of sharetokens and round-robin scheduling gives proportional share scheduling, and the fact that sharetokens are evenly spaced in the run queue ensures that the CPU is allocated among processes with minimal burstiness or jitter.

Second, the waking queue is used to support processes with explicit deadlines. As Appendix A demonstrates, these processes are guaranteed to meet their deadlines as long as the CPU is not overloaded (as per EDF [Liu 73]). Furthermore, the proportional share mechanism simultaneously ensures that the amount of resource given to periodic deadline-driven processes continues to be proportionate in overload, and that non-deadline-driven processes cannot starve under these conditions.

Third, the other role of the waking queue is to support ‘deferred processing’ for processes that have voluntarily blocked during their last quantum. The purpose of this is to favour I/O bound processes over CPU bound processes according to the traditional workstation scheduling approach. The waking queue scheme also enables a significant optimisation which reduces the number of times that *unblock()* and *block()* must be called when processes voluntarily block while being scheduled from the waking queue.

And, finally, the accumulated time mechanism is used to support deferred processing for processes that have missed multiple sharetoken-turns and so allows I/O bound processes to maintain their fair share of CPU time over the long term.

A significant area of planned future development is the integration of our SHRED implementation into the user-level scheduler framework reported in [Coulson 01]. This framework employs ‘virtual processors’, which we are implementing as SHRED-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 SHRED to be a key factor in enabling this platform to provide more predictable and adaptive quality of service to its applications.

REFERENCES

- [Banachowski 02] Banachowski S.A., Brandt S.A., *The BEST Scheduler for integrated processing of best-effort and soft-real-time processes*. Multimedia Computing and Networking 2002 (MMCN '02) –January 2002
- [Beck 98] Beck, M., Bohme, H., Dziačka, 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.
- [Duda,99] Duda, K.J., Cheriton, D.R., “Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler”, Proc. 17th ACM Symposium on Operating Systems Principles, ISSN:0163-5980, pp 261-276, 1999.
- [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., Hanco, 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.

[Steinmetz 02] Steinmetz R., Nahrstedt K., *Multimedia Fundamentals – Volume 1: Media Coding and Content Processing* – Prentice Hall 2002.

[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.

APPENDIX A: ACCOMMODATING PERIODIC DEADLINE-DRIVEN PROCESSES

In this appendix, we analyse the important issue of our scheduler’s treatment of periodic deadline-driven processes. In particular, we analyse the behaviour of these processes and prove that they *always meet their deadlines in underload conditions*: it is straightforward and intuitive to see that SHRED 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 n sharetokens such that $n/x = C/T$, where x 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 ⁷.

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 $(x/n) - 1$ non-P sharetokens, i.e. $((x/n) - 1)q$ ms, have elapsed. During that interval it would nominally require $((x/n) - 1)NC$ ms of processing time. Now, $((x/n) - 1)NC = ((T/C) - 1)NC = (T - C)N$, which is precisely the time that P was

⁷ It is possible in our scheme that a periodic deadline-driven process may miss its *first* deadline. For example, referring to the example in figure 1, if the process being added (with 5 sharetokens) has $C = 10$ ms and $T = 26$ ms, and $q = 10$ ms, the first deadline of 26ms will clearly be missed. However, the analysis below demonstrates that all subsequent deadlines will be honoured.

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 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 x/n sharetokens apart, we can deduce the fact that $NT = (x/n)q$.

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 $NT = (x/n)q$, we infer that P will have enough processing time in the waking queue provided that $C/T \leq (n/x)$, 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 always meet their deadlines.

APPENDIX B: ALGORITHMS FOR BLOCKING AND UNBLOCKING

This appendix presents in detail the algorithms required to block and unblock a process—i.e. to remove/ add its sharetokens from/ to the run queue. Please refer to section 2.2 for context and orientation.

The required data structure is as follows:

```
typedef struct st {
    Process owner;
    int index;
    Sharetoken *rsucc; /* run queue */
    Sharetoken *rpred; /* run queue */
    Sharetoken *snext; /* static queue */
    Sharetoken *pnext; /* per-proc queue */
    ...
} Sharetoken;

/* pointer to front of runqueue */
Sharetoken *runqueue;
```

The *owner* field in the *Sharetoken* struct refers to the process that owns this sharetoken; the *index* field records the position (0.. n) of the sharetoken in the static queue; the *rsucc* and *rpred* fields support the dynamic run queue as a doubly-linked list; and the *snext* field supports the static sharetoken queue as a singly-linked list. We also employ an additional queue called the *per-process queue* that holds all the sharetokens belonging to a particular process. Sharetokens participate in this queue (which is implemented as a singly-linked list) using the *pnext* field.

We also assume that the process descriptor struct (called *Process*) includes the following fields: i) a pointer called *stqueue* that points to the front of the process’s *pnext* list; ii) a boolean called

runnable that states whether or not the process is currently runnable.

Given the above, the algorithms for blocking a process (i.e. removing it from the run queue), and unblocking a process (i.e. restoring it to its former position in the run queue) are as follows:

```
block(Process proc)
{
  Sharetoken* p = proc->stqueue;
  proc->runnable = FALSE;
  while (p) {
    if (p == runqueue) {
      /* need to update front of runqueue */
      runqueue = (p->rsucc==p) ? NULL : p->rsucc;
    }
    p->rpred->rsucc = p->rsucc;
    p->rsucc->rpred = p->rpred;
    p = p->pnext;
  }
}
```

```
unblock(Process proc)
{
  Sharetoken *p = proc->stqueue;
  Sharetoken *r = runqueue;
  Sharetoken *frontp, front; /* sentinel */

  proc->runnable = TRUE;
  frontp = &front /* sentinel */
  frontp->rpred = frontp->rsucc = NULL;
  while(p != NULL && r != NULL) {
    if (p->index < r->index) {
      next = p; p = p->pnext;
    } else {
      next = r; r = r->rsucc;
    }
    next->rsucc = newp->rsucc;
    next->rpred = newp;
    newp->rsucc->rpred = next;
    newp->rsucc = next;
  }
}
```

```

}
while(p != NULL) {
  /* nothing left on r list */
  p->rsucc = newp->rsucc;
  p->rpred = newp;
  newp->rsucc->rpred = p;
  newp->rsucc = p;
  p = p->pnext;
}
while(r != NULL) {
  /* nothing left on p list */
  r->rsucc = newp->rsucc;
  r->rpred = newp;
  newp->rsucc->rpred = r;
  newp->rsucc = r;
  r = r->rsucc;
}
/* tidy up front of new run queue */
frontp->rsucc->rpred = NULL;
runqueue = frontp->rsucc;
}
}
```

The *block()* algorithm simply iterates over the blocking processes' sharetokens and successively removes each one from the run queue (while leaving it on the static queue). The *unblock()* algorithm uses the index field, which reflects the ranking of the sharetokens in the static queue, as the basis on which to merge the process' sharetokens into the run queue.