The following paper was originally published in the
Proceedings of the USENIX Annual Technical Conference (NO 98)
New Orleans, Louisiana, June 1998

The Eclipse Operating System: Providing Quality of Service via Reservation Domains

John Bruno, Eran Gabber, Banu Ozden, and Abraham Silberschatz
Bell Laboratories, Lucent Technologies

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

The Eclipse Operating System: Providing Quality of Service via Reservation Domains

John Bruno, Eran Gabber, Banu Özden and Abraham Silberschatz
Bell Laboratories, Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07974
{jbruno, eran, ozden, avi}@research.bell-labs.com

Abstract

In this paper, we introduce a new operating system abstraction called *reservation domains*, and describe its implementation in Eclipse, an experimental operating system that provides a testbed for Quality of Service (QoS) support for applications.

Reservation domains enable explicit control over the provisioning of system resources among applications in order to achieve desired levels of predictable performance. In general, each reservation domain is assigned a certain fraction of each resource (e.g., 25% CPU, 50% disk I/O, etc.). Eclipse implements reservation-domain scheduling of multiple resources. It currently supports CPU and disk and physical memory (working set size) scheduling.

Eclipse implements a new scheduling algorithm, Move-to-Rear List Scheduling (MTR-LS), that provides a *cumulative service guarantee*, in addition to fairness and delay bounds. Cumulative service guarantee is necessary for ensuring predictable aggregate throughput for applications that require multiple resources. Preliminary experiments indicate that MTR-LS provides good QoS in overloaded systems. In particular, MTR-LS favors *less-greedy* processes.

The Eclipse operating system is based the Plan9 from Bell Labs, and can run any Plan9 application without modification. Eclipse emphasizes the use of per-process name space, and it can schedule any I/O device or user level file system without any change to device driver or file system code.

1 Introduction

New multimedia applications, which require support for real-time processing, are pacing the demand for operating system support for Quality of Service (QoS) guarantees. The desire to support multiple real-time applications on a single platform requires that the operating system have the ability to provision system resources among applications in a manner that achieves the desired levels of predictable performance. Moreover, computer networks are starting to provide QoS guarantees with respect to packet delay and connection bandwidth. These QoS guarantees are of little use if they cannot be extended to applications executing at the endpoints. Current general-purpose multiprogrammed operating systems do not provide QoS guarantees since the performance of a single application is, in part, determined by the overall load on system. As a result, many users prefer to use stand-alone systems with limited dependency on shared servers in order to achieve some semblance of QoS by indirectly controlling the system workload. Real-time operating systems are capable of delivering performance guarantees such as delay bounds, but require that applications be modified to take advantage of the real-time features.

Our goal is to provide QoS guarantees in the context of a general-purpose multiprogrammed operating system, without modification to the applications, by giving the user the option to provision system resources among applications in order to achieve the desired performance levels.

This paper introduces a new operating system abstraction called *reservation domains*, which is intended to provide predictable QoS by controlling resource provisioning. The basic idea behind reservation domains is to isolate the performance of reservation domains from each other. In particular, time-sensitive processes can coexist with batch processes on the same system.

We discuss the importance of *cumulative service* guarantee, which complements other QoS parameters such as *delay* and *fairness*. Informally, guaranteed cumulative service means that the scheduling delays encountered by a process on various resources do not accumulate over the lifetime of the process. In other words, a process that is competing for resources will execute at a predictable rate, which is determined by the fraction of the resources that is reserved for that process, regardless of the intensity of the competition for the resource.

We continue with the description of a new schedul-

ing policy called Move-To-Rear List Scheduling (MTR-LS), that provides cumulative service guarantees, as well as fairness and delay bounds.

Next we describe the Eclipse operating system, which is a testbed for operating system support for QoS. Eclipse implements reservation domains and MTR-LS scheduling. Eclipse can schedule multiple resources independently. The current implementation can schedule CPU, disk I/O and physical memory (working set). Experimental results indicate that MTR-LS outperforms the standard Plan9 priority scheduler and a weighted round-robin scheduler. We also include an excerpt from the Eclipse implementation of MTR-LS, which illustrates the simplicity of the implementation.

The remainder of this paper is organized as follows. In Section 2 we discuss related work. Section 3 introduces the concept of a reservation domain. Section 4 presents the QoS parameters of interest. Section 5 presents the MTR-LS scheduling policy and its properties. The implementation of the Eclipse operating system is described in Section 6. Section 7 presents the results of our preliminary experiments with Eclipse. Finally, Section 8 describes our future work and conclusions. The Appendix contains a code excerpt from the Eclipse implementation of MTR-LS.

2 Related Work

Stride scheduling [16] and lottery scheduling [17] attempt to provide each process with a share of the server in proportion to its corresponding weight (number of "tickets"). Start-time fair queuing [4] and earliest eligible virtual deadline first [13] guarantee fairness bound. Class based queuing (CBQ) [15] uses bandwidth reservations to schedule packets on shared data links. Processor capacity reserves [7] provides each process with its reserved share and delay guarantees. The SMART scheduler [8] provides predictable performance for real-time applications, which may executed concurrently with non real-time applications. SMART allows processes to specify their scheduling constraints, and provides dynamic feedback to applications when the constrains cannot be met. The Rialto operating system [5] provides resource reservations similar to Eclipse. However, both SMART and Rialto schedule only CPU. The Nemesis operating system [6] provides accurate accounting of resource consumption by running the application and most of the kernel code in the same address space. Nemesis is similar to Eclipse by providing predictable performance via allocation of CPU and disk I/O to domains. However, Nemesis employs a radically different OS structure, which necessitates rewriting of most applications and device drivers. The above algorithms and systems were not designed with our cumulative service measure in mind, so it is not surprising that the properties they do enjoy are not sufficient to guarantee cumulative service effectively.

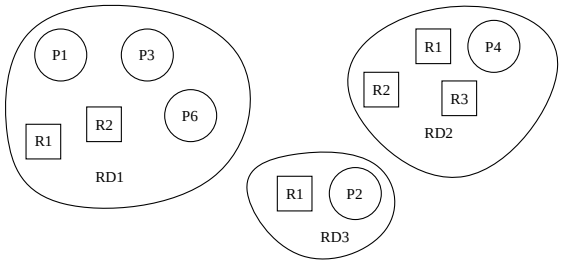

Figure 1: A Reservation-Domains System

3 Reservation Domains

In order to incorporate QoS into operating systems, we introduce the notion of a *reservation domain*. A reservation domain is a collection of processes and corresponding resource reservations. A computer system may run several reservation domains and provide several types of resources (e.g., CPU, disk, network, physical memory), which are reserved and scheduled independently. The processes that belong to a particular reservation domain are guaranteed to receive at least their reserved portions of the domain's associated resources. Figure 1 illustrates a system that runs three reservations domains: RD_1, RD_2 and RD_3. Each domain contains an explicit resource reservation for the resources $R1$, $R2$ and $R3$. One or more processes may run within a single reservation domain.

Reservation domains are designed to combine the advantages of real-time, time-sharing, and stand-alone systems. Benefits of reservation domains, which are impossible to achieve with priority based scheduling or real-time scheduling algorithms, are:

- Provides QoS guarantees even when the system is overloaded. In fact, a reservation domain is similar to a smaller, dedicated machine. Application programs need not be rewritten to use real-time services in order to deliver predictable QoS in a shared environment.
- Allows division of resources according to a policy. For example, two reservation domains may each reserve half the CPU, although one of them contains more processes than the other, and all processes are CPU bound.
- Supports interesting ways of controlling the computing environment. For example, the windowing system may be able to change resource reservations of domains associated with the window in focus (important) and of closed windows (less important). Another possibility is a resource super-

visor, that adapts resource reservations dynamically according to the "complaints" of processes. A process that misses its deadlines will complain, which will prompt the supervisor to increase its reservations. A process that meets its deadlines will keep silent, and will get a smaller reservation.

A variety of applications can benefit from reservation domains. For example, soft-real-time applications can use bounded response times, hosting applications can benefit from provisioning the system resources, and long running OLAP (on-line analytical processing) applications can use protection between reservation domains to complete their tasks in a timely manner without impacting current workload. Reservation domains can also benefit PC users, who would like to run several applications concurrently, when the applications are written with the assumption that they run on a dedicated machine.

4 QoS Guarantees

The reader who is familiar with QoS parameters and with [2] may skip this section. In this section we introduce three Quality of Service (QoS) parameters: cumulative service, delay bound and fairness. Cumulative service is a new QoS parameter, while delay bound and fairness are well known, and many scheduling algorithms do provide them. We start with an example that shows the need for the cumulative service guarantee. The definition of fairness and delay bounds will be given at the end of this section.

Example 1: We assume that the system runs one I/O bound process together with n infinite loops. The system employs a weighted round-robin scheduler, similar to the one described in Section 6.2. The I/O bound process issues I/O requests sequentially. It requires 1 msec. of CPU in order to issue the next I/O request. Each I/O request requires 23 msec. to complete. There is no contention on the I/O device. We reserve 0.5 of the CPU to the I/O bound process. We expect that each iteration of the I/O bound process will last $2 + 23$ msec., where 2 msec. is the expected execution time of the CPU phase of each iteration (1 msec. on an idle machine is equivalent to 2 msec. elapsed time when the process is reserved 0.5 of the CPU). Thus the expected execution rate of this program is $1000/25 = 40$ iterations per second.

However, whenever the I/O bound process arrives at the CPU, it is appended to the end of the ready processes queue, so it has to wait for at least n time slices before it can run. Thus the actual execution time of each iteration is $1 + n\delta + 23$, where δ is the time slice length. For example, for $n = 10$ and $\delta = 10$ msec. the execution rate of the I/O bound process is $1000/(1+100+23) \approx 8$, which is one fifth of the expected rate!

Note that the round-robin scheduler provides fairness and delay bounds, since all processes are scheduled within $n + 1$ time slices. However, the execution rate of the I/O bound process can be arbitrarily low, since round-robin does not provide a cumulative service guarantee. □

We will need the following definitions for the rest of the section: A system is considered to be a collection of resources (servers). Each resource is modeled by a "service rate" and a "preemption interval" Δt. Δt is the minimum time that the resource must run before a preemption can occur. A resource with a zero preemption interval can preempt a process at any time.

A process is considered to execute an ordered set of of *phases*, where a phase is a resource-duration pair, (s, t), where s is one of the system resources and t is the amount of time it would take resource s to complete the phase running alone on the resource. The phases of a process are not known in advance. The identity of the next phase is known only after executing the previous phase.

Even though we are interested in the performance of our system over all resources, it turns out that, due to the definition of the cumulative service guarantee, it is sufficient to study the performance at a single resource. From the point of view of resource s, a process is denoted by an ordered set of phases that alternate between resource s and *elsewhere*. The "elsewhere" resource represents the phases of processes at resources other than s.

The motivation for reservation domains is to isolate the each reservation domain from the others. We would like to guarantee a lower bound on the performance of a reservation domain, which is independent of other reservation domains. Let α_i be the fraction of a resource allocated to a reservation domain D_i. Ideally, each reservation domain D_i should receive at least α_i fraction of the resource whenever D_i has a process requiring the resource (a.k.a., whenever D_i is *busy*). We refer to the minimum service time received in the *idealized* service model as *virtual service time*. We denote the virtual service time received by reservation domain D_i in any real time interval $[\tau, t]$ by $v_i(\tau, t)$. Similarly, we denote the *real service time* (running on the resource) received by reservation domain D_i in any real time interval $[\tau, t]$ by $s_i(\tau, t)$. Note that $\alpha_i v_i(\tau, t) = s_i(\tau, t)$.

Realizable scheduling policies require that we run at most one process at a time on the resource. This means that if there is more than one process waiting to run on the resource, then one or both of the processes will experience (queuing) delay. We define $w_i(\tau, t)$ be the cumulative *real waiting time* (blocked by other domains' processes running on the resource) obtained by domain D_i in the interval $[\tau, t]$. By definition, $w_i(\tau, t) + s_i(\tau, t)$ is the total *real time* spent by domain D_i at the resource

in the interval $[\tau, t]$ either by running or waiting.

In the idealized model, receiving $v_i(\tau, t)$ virtual service time takes at most $v_i(\tau, t)$ real time. With realizable scheduling policies, in order to provide a performance as good as the one in the idealized model, the total real time $w_i(\tau, t) + s_i(\tau, t)$ spent by domain D_i at the resource in the interval $[\tau, t]$ to receive $v_i(\tau, t)$ virtual service time should be less than or equal to $v_i(\tau, t)$. We express the real system's ability to match the performance of the idealized system in terms of a cumulative service guarantee [2].

Definition 1: We say that a scheduling policy provides a *cumulative service guarantee* if there exists a constant K such that for all domains D_j and $\tau \leq t$, we have $v_j(\tau, t) \geq w_j(\tau, t) + s_j(\tau, t) - K$. □

Although the definition of cumulative service guarantee is in terms of a single resource, it implies a "global" cumulative service guarantee (using cumulative virtual service time and cumulative real time over all resources) in the multi-resource case where there is a constant number of resources [2]. Guaranteeing cumulative service is vital for applications that require multiple resources and arrival of a phase on a resource depends on the departure of previous phases. Cumulative service guarantee is necessary to ensure a predictable aggregate throughput over all the resources for such applications.

We will define the delay guarantee for the case when at any given time there is only one process within a reservation domain. For the more general case, when there can be multiple concurrent processes within a domain, the delay a phase experiences also depends how the reservation domain schedules its phases. This case requires elaborate treatment, and will not be covered in this paper.

Definition 2: A scheduling policy provides *delay bound* if, for a phase of any domain D_j, the real waiting time plus service time to complete the phase takes at most a constant amount more than d/α_j, where d is the duration of the phase. □

The fairness parameter measures the ability of the system to ensure that domains that are simultaneously contending for the same resource will "share" that resource in *proportion* to their reservations, independent of their previous usage of the resource [3]. That is, a fair scheduling policy does not penalize a domain that utilized an idle resource beyond its reservation when other domains become busy on that resource.

The definition of fairness is based on another idealized service model called *processor sharing* [11]. Under processor sharing, each domain receives a service proportional to its fraction on a resource. Since ideal processor sharing cannot be implemented in practice, fairness is defined as:

Definition 3: A scheduling policy is *fair* if there exists a constant F such that for any time interval $[\tau, t]$ during which a pair of domains, D_i and D_j, both continuously require the resource, we have $|s_i(\tau, t)/\alpha_i - s_j(\tau, t)/\alpha_j| \leq F$. □

Reservation domains specify their QoS requirements by providing a service fraction α_i for each system resource. Admission control ensures that the sum of the service fractions of all domains do not exceed certain prescribed limits. Admission control is necessary for delay and cumulative service guarantees. It is not required for fairness.

5 Move-to-Rear List Scheduling

This section presents the Move-To-Rear List Scheduling (MTR-LS) policy, which provides a cumulative service guarantee, is fair, and has bounded delay. MTR-LS policy is a generalization of the one presented in [2]. The main difference is that we allow here the aggregate quantum allocated for a domain to be partitioned. This improves QoS guarantees and average delay and waiting time. In the following subsections we present the MTR-LS policy, its complexity, and its properties.

5.1 Algorithm and Data Structures

Central to the MTR-LS policy is an ordered list, $\mathcal{L}$, of pairs *(i,left)* where i is the index of a reservation domain (i.e., D_i) and *left* is size of the *quantum*, which is the maximal amount of service time reservation domain D_i can receive without being interrupted. There can be multiple occurrences of domain D_i on $\mathcal{L}$, that is, pairs appearing at different positions in list $\mathcal{L}$ which have the same first coordinate i. The pairs *(i,left)* will be called *tokens* in the following discussion.

We say that domain D_i is before D_j on $\mathcal{L}$ if the first token of domain D_i on $\mathcal{L}$ appears before all tokens of D_j on $\mathcal{L}$. Each domain, whether it is busy or idle, has at least one token on $\mathcal{L}$. The system defines the *service cycle* as the constant T, which is the upper bound on the sum of all the quanta represented in list $\mathcal{L}$. The sum of quanta in $\mathcal{L}$ that belong to domain D_i is equal to $\alpha_i T$. We require that $\alpha_i T$ must be an integer.

MTR-LS performs the procedure shown in Figure 2 at every *decision epoch*. Decision epochs occur at the time of the arrival of a new domain, the departure of a domain, the expiration of the current quantum, the completion of the phase of the current running process (e.g. the process blocks), or the end of the current preemption interval. In other words, a currently running process may be preempted only at the end of the current preemption interval (Δt) or at the end of the current quantum.

Figure 3 depicts the domain update routine, which is called every decision epoch to update the token of currently running domain. This routine will either split the current token into two, leaving one token in the cur-

Decision_Epoch()
stop **elapsed** counter;
if a new domain D_i is admitted to the system **then**
append the token $(i, \alpha_i T)$ to list $\mathcal{L}$;
if a domain D_i is removed from the system **then**
remove all tokens belonging to D_i from $\mathcal{L}$;
if state == **busy then**
Update_Domain();
Run_a_Domain();

Figure 2: Decision Epoch Processing

rent place and appending the other to the end of $\mathcal{L}$, or move the token entirely to the end of $\mathcal{L}$, depending on the remaining quantum in the token. In any case, the sum of the quanta in the two tokens is equal to the quantum in the original token. Figure 4 shows the routine Run_a_Domain, which selects the next domain to run on the resource. The counter **elapsed** records the elapsed time to the next decision epoch.

The service obtained by a process may be less than the allocated quantum due to the termination of a phase or the arrival of a process. In the former case, the phase terminates, the process goes elsewhere, and the first runnable domain on $\mathcal{L}$ is serviced next. In the latter case, if the arriving process belongs to a domain which is ahead of the current running process's domain in the list $\mathcal{L}$, then the running process is preempted (as soon as the preemption interval permits) and a process of the first runnable domain on $\mathcal{L}$ is serviced next.

Figure 5 shows the Combine_Elements routine, which combines neighboring tokens belonging to the same domain. It is called whenever the list $\mathcal{L}$ is changed. This routine has to examine $\mathcal{L}$ only in the vicinity of the last change (removal or addition of tokens).

Example 2: Figure 6 illustrates the operation of MTR-LS. There are three tokens on $\mathcal{L}$, which belong to domains D_1, D_2 and D_3. The processes P_1, P_2 and P_3 are associated with domains D_1, D_2 and D_3, respectively. The total quanta of D_1, D_2 and D_3 are 10, 5 and 15, respectively. Initially (Figure 6a) all the domains are busy. Since the first busy domain is D_1, process P_1 is executed. P_1 goes "elsewhere" after receiving seven time units of service. $\mathcal{L}$ is updated such that quantum of D_1

Update_Domain()
Let *(i,left)* be the current token being serviced in $\mathcal{L}$
Append *(i,***elapsed***)* to the end of list $\mathcal{L}$;
left' ← *left* - **elapsed**;
if *left'* == 0 **then**
remove current token *(i,left)* from $\mathcal{L}$
else
replace current token with *(i,left')*;
Combine_Elements();

Figure 3: Domain Update

Run_a_Domain()
if there is no runnable domain on the list $\mathcal{L}$ **then**
state ← **idle**;
else
Let *(j,left)* be the first runnable token in $\mathcal{L}$;
state ← **busy**;
run D_j on the resource for at most *left* time units (current quantum);
start **elapsed** timer;
wait for next decision epoch;

Figure 4: Run a Domain

is partitioned as shown in Figure 6b. Now the first busy domain is D_2. Therefore, process P_2 is executed. Once the quantum of D_2 is exhausted in five unit of time, D_2's token is placed at the tail of the list (Figure 6c). The next busy domain is D_3. Therefore, process P_3 selected for execution. After three units of time, domain D_1 becomes busy (process P_1 arrives at the resource). Since D_1's token precedes D_3's token in the list, P_3 is preempted. The quantum of D_3 is partitioned as shown in Figure 6d, and process P_1 is scheduled. □

5.2 Complexity

In a simple implementation of MTR-LS, a domain may have multiple tokens on $\mathcal{L}$. Each token corresponds to a quantum of a given size and the sum of all the quanta for a given domain is equal to $\alpha_i T$. The list $\mathcal{L}$ should support the following operations: 1) find the first busy domain in $\mathcal{L}$; 2) split a token, and append one of the parts at the end of $\mathcal{L}$; the original token is either updated in place or deleted; 3) append tokens of a new domain; 4) delete all tokens of a domain; and 5) combine tokens in $\mathcal{L}$. All of the operations except the first may be performed in constant time. The first operation may take $O(T)$ if all domains have $\alpha_i T$ tokens of one time quantum each.

The worst case running time of MTR-LS may be improved by keeping the domains in a heap instead of in a list, marking the tokens by an increasing time-stamp, and sorting the domains by the lowest time-stamp they currently have. The complexity of this implementation is $O(\log n)$ where n is the number of busy domains. Further time reduction can be achieved by using a priority queue or other data structures that are described in [14].

5.3 Properties of the MTR-LS Policy

The MTR-LS policy provides fairness, delay bound and cumulative service guarantees. The proofs for these properties can be found in in [1]. In particular, MTR-

Combine_Elements()
if (i, m_1) and (i, m_2) are consecutive in $\mathcal{L}$ **then**
replace them with $(i, m_1 + m_2)$

Figure 5: Combines tokens in $\mathcal{L}$

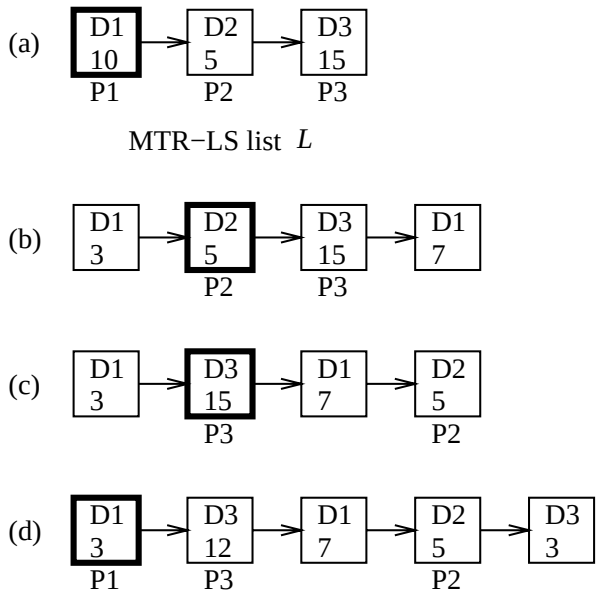

Figure 6: Example of MTR-LS Scheduling

LS policy is fair with a bound of T. MTR-LS provides cumulative service guarantee with a bound of T if $\sum_i \alpha_i \leq 1$ and the preemption interval is zero, which means that a running process is preempted immediately by a ready process that is ahead of it in $\mathcal{L}$. However, if the preemption interval is positive, MTR-LS requires that $\alpha_j \leq 1 - \sum_i \alpha_i$ for all domains D_j in order to provide a cumulative service guarantee with bound T. The reason is that a ready process may be delayed by the currently running process until the end of the current preemption interval. The delay bound provided by MTR-LS follows directly from the cumulative service guarantee (Section 4).

6 The Eclipse Operating System

Eclipse is derived from the Plan9 operating system from Bell Labs [12]. It provides a per-process name space, access to resources via the name space, a file access protocol (9P), that provides a uniform interface to all servers, and many interesting file servers, such as the windowing system $8\frac{1}{2}$, an FTP file system, etc. [10].

Eclipse provides reservation domain functionality on top of Plan9, without any change to existing applications or servers. Eclipse is compatible with Plan9; that is, Eclipse retains the external interface of Plan9 (system calls, protocols, name space structure, etc.).

Eclipse implements reservation and scheduling of CPU, I/O and physical memory (working set size), which we describe in Sections 6.2, 6.3, and 6.4, respectively. The resources are managed independently. Resource reservations and reservation domain management can be done by the shell; no systems level programming is needed.

name	access	description
`rdcpu`	r/w	% CPU reservation of current RD
`rdcpusum`	r	sum of current CPU reservations
`rdcpulim`	r	maximum allowed CPU reservation
`rdio`	r/w	% I/O reservation of current RD
`rdiosum`	r	sum of current I/O reservations
`rdiolim`	r	maximum allowed I/O reservation
`rdmem`	r/w	physical memory reservation in KB
`rdmemsum`	r	sum of physical memory reservation
`rdmemlim`	r	available physical memory
`rdsched`	r/w	current CPU scheduling algorithm

Table 1: Control Files for Resource Reservation

Since the sum of the explicit reservations of a resource $\sum_i \alpha_i$ may be less than one, Eclipse distributes the unreserved portion of the resource evenly among all reservation domains. For example, the effective reservation of domain i is $\alpha_i + (1 - \sum \alpha_i)/n$, where n is the number of reservation domains.

6.1 Managing Reservation Domains

Eclipse maintains a list of all active reservation domains (RDs). Each process belongs to a single reservation domain. A new process inherits the reservation domain of its parent, unless the process is created by `rfork(RFPROC|RFRDG)`, which places the process in a newly created reservation domain. A process may remove itself from its current reservation domain and start a new reservation domain by an appropriate call to `rfork`. In fact, each process may belong to a different reservation domain. A new reservation domain is created without any explicit reservations. However, the new domain is assigned an effective reservation, that includes its proportional share of the unreserved portion of the resources, as described above.

Resource reservation is accomplished by writing strings to the appropriate control files. Table 1 describes some of the control files. For example, the shell command `echo 50 > /dev/rdcpu` requests reservation of 50% of the CPU to the reservation domain which runs this command. Eclipse will deny resource reservations if total reservation exceeds the corresponding limit. Eclipse deletes a reservation domain and releases its resources as soon as the last process belonging to this reservation domain is terminated.

6.2 CPU Scheduling

Eclipse implements MTR-LS and weighted round-robin scheduling (WRR), which can be selected by the `rdsched` control file. For example,

`echo w > /dev/rdsched` selects WRR, and `echo m > /dev/rdsched` selects MTR-LS.

Weighted Round-Robin (WRR) is similar to round-robin scheduling, but the CPU slice of each process is proportional to its reservation. In our case, the CPU slice of the processes belonging to a particular reservation domain is proportional to the domain's reservation. Note that WRR scheduling without an explicit CPU reservation is identical to regular round-robin.

Eclipse implements WRR scheduling by maintaining a a *cpu_time_left* field for each reservation domain, which contains its remaining CPU reservation for the current *CPU service cycle*. Eclipse maintains two ready queues: *run_hi* and *run_lo*. Ready processes are appended to *run_hi* if the corresponding reservation domain has a *cpu_time_left* > 0. Otherwise, the process is appended to *run_lo*. Eclipse selects the first process from *run_hi* that has an associated *cpu_time_left* > 0. Requests with *cpu_time_left* ≤ 0 are moved to *run_lo*. The clock interrupt handler decreases the corresponding *cpu_time_left*, appends the current process to either *run_hi* or *run_lo* according to the above criterion, and calls the scheduler. The *cpu_time_left* of all reservation domains is renewed at the beginning of every CPU service cycle according to the corresponding CPU reservation. At this time all processes are moved from the *run_lo* to the *run_hi* queue.

Eclipse does not maintain a global tokens list $\mathcal{L}$ for MTR-LS, rather it keeps the creation time-stamp for each token, and sorts the reservation domains by the lowest time-stamp of the tokens that they hold. In this way, the list $\mathcal{L}$ is replaced with a collection of cyclic buffers, one for each reservation domain, which keep the tokens of that domain. Each reservation domain consumes its tokens in their time-stamp order. The Appendix contains code excerpts from the Eclipse kernel that illustrate the implementation of MTR-LS. This implementation scans the ready process list, and selects the ready process that is associated with the reservation domain that holds the token with with the lowest time stamp. The advantage of this implementation is its simplicity. However, the time complexity of each scheduling decision is $O(n)$, where n is the number of ready processes. A more sophisticated implementation may scan the ready domains queue instead, which may reduce the scheduling time. The overhead of maintaining the tokens during a clock interrupt is constant.

The current Eclipse implementation uses clock interrupts rate of 200Hz. The CPU service cycle (T) is 1/2 second long. More frequent clock interrupts improve responsiveness at the expense of more frequent context switches.

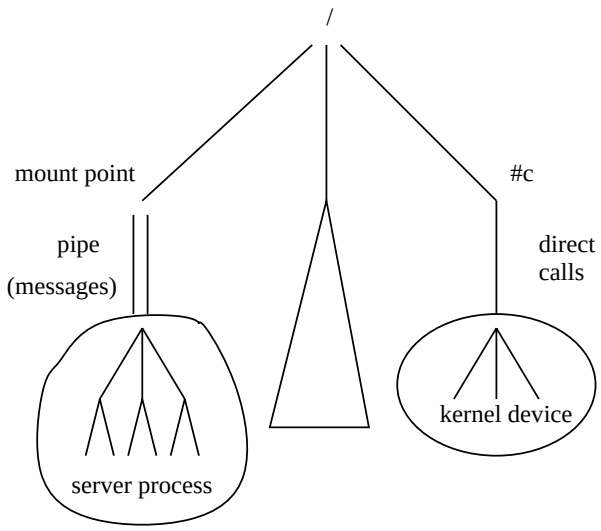

Figure 7: Eclipse/Plan9 Name Space

6.3 I/O Scheduling

Eclipse assumes that I/O devices handle a series of non-overlapping requests (one request at a time), although servers may queue several requests internally. This assumption is adequate for disks with single arms, but grossly inaccurate for network devices, which can handle several full duplex streams of interleaved packets.

The implementation of I/O scheduling is closely related to the structure of Eclipse/Plan9 name space, as illustrated in Figure 7. The name space is formed by joining independent hierarchies of files, which are provided by file servers and kernel devices. File servers are user level processes, which respond to 9P requests over bi-directional pipes. Kernel drivers are directly called by the kernel, which avoids the 9P protocol overhead.

Only a part of the name space may be subject to I/O scheduling, according to a flag in the `mount` and `bind` system calls. All other parts of the name space are not affected by I/O scheduling. This is essential, since all resources in Eclipse are part of the name space. For example, it is inappropriate to schedule requests to screen and mouse. Moreover, an attempt to schedule paging requests may cause a deadlock.

Eclipse implements I/O scheduling by intercepting `read` and `write` system calls, as illustrated in Figure 8. The rectangles inside each reservation domain denote resource reservations, and the circles denote processes. Since all I/O requests pass through the same system calls interface, I/O scheduling can be applied to all clients and all servers, without any change to either. Moreover, clients and servers may not be aware that they are subject to I/O scheduling. Scheduling can be applied to both file servers and kernel devices with equal ease, without any change to them.

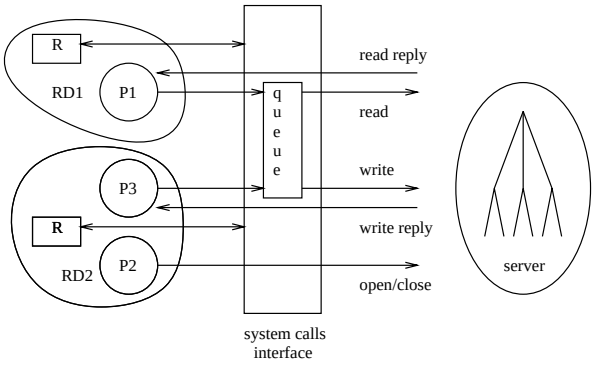

Figure 8: Implementation of I/O Scheduling

I/O scheduling is implemented similar to WRR CPU scheduling, by maintaining two I/O queues, *io_hi* and *io_lo*, and an *io_time_left* field for each reservation domain, which is reduced by the elapsed time of the request. Since the server may re-order the requests, the elapsed time is computed from the end of the previous request (if the previous request ended after the current request was delivered to the server). The reservations are renewed at the beginning of every I/O service cycle, and the requests are moved to the *io_hi* queue.

Eclipse allows the server to queue up to *maxactive* requests internally, since performance is generally improved when servers have some internal queuing (e.g., by sorting disk accesses to reduce arm movement). The current implementation has a single global requests queue. Future versions will have a queue for each server. An I/O service cycle is 1/2 second long and *maxactive* is 4.

The above scheduling algorithm is bypassed if the server is currently idle or if there is no contention (all active requests belong to the same reservation domain).

6.4 Memory Scheduling

Eclipse implements reservation of physical memory by a selective page-out strategy. The pager process scans the entire memory, and selects pages which where not accessed since the last scan and belong to reservation domains that exceeded their reservation of physical memory. This strategy ensures that the working set size does not fall below the reservation, and all pages above the reservation are subject to global LRU replacement.

6.5 Source Code

Eclipse is derived from the 2nd edition of Plan9. It is currently implemented on PC based machines. Table 2 describes the breakup of source lines in Plan9 and Eclipse. In both cases, we refer to an extravagant kernel including all device drivers and protocols. As Table 2 indicates, Eclipse is machine independent. The tiny machine dependent part of Eclipse is contained in the clock interrupt handler, and is shown in the Appendix. It can be easily ported to other architectures.

part	Plan9 2nd Ed.	Eclipse
portable	31,046 lines	$\pm$1,143 lines
PC dependent	19,031 lines	+22 lines
total	50,077 lines	51,040 lines

$\pm$ means changed source lines (additions and deletions)

Table 2: Plan9 and Eclipse Source Size

7 Experience and Measurements

Eclipse has been operational since October 1995. It proved to be stable, useful and instructional. In particular, we can easily demonstrate and control extreme conditions, such as overloading, memory thrashing and starvation. We use Eclipse for running multimedia applications, such as MPEG players, concurrently with other demanding activities. The applications meet their deadlines when they run in reservation domains with appropriate resource reservations.

We used two methods for determining the appropriate reservations: manual trial and error (actually, a binary search), and an automatic resource monitor, which is a user-level process, that collect "complaints" from application programs and adjusts the reservations appropriately. The automatic monitor increases the reservation (fast) when the application suffers from delays, and reduces the reservation (slowly) when the application is not delayed.

We have two MPEG-1 players for Eclipse: a player which reads from a file, and a $\mathcal{F}ellini$ [9] client, which receives a stream of information from the $\mathcal{F}ellini$ continuous media server.

The rest of this section describes several experiments that illustrate the effectiveness of MTR-LS scheduling algorithm. In those experiments we used reservation domains extensively to distinguish between several classes of applications that run concurrently on the same machine.

7.1 Experimental Setup

All experiments were run on a Micron Millenia PC with an Intel Pentium processor running at 133MHz. The PC has a Talisman XL MPEG-1 decoder card.

The experiments compared three scheduling algorithms: the original Plan9 scheduler, which is a priority based scheduler, that adjusts the priorities of processes according to their CPU consumption, the MTR-LS scheduler, and a weighted round-robin (WRR) scheduler.

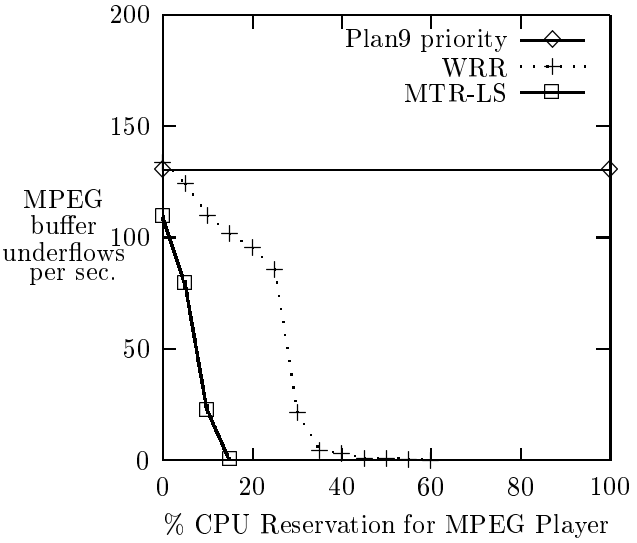

Figure 9: Buffer Underflows of an MPEG Player Under CPU Overload

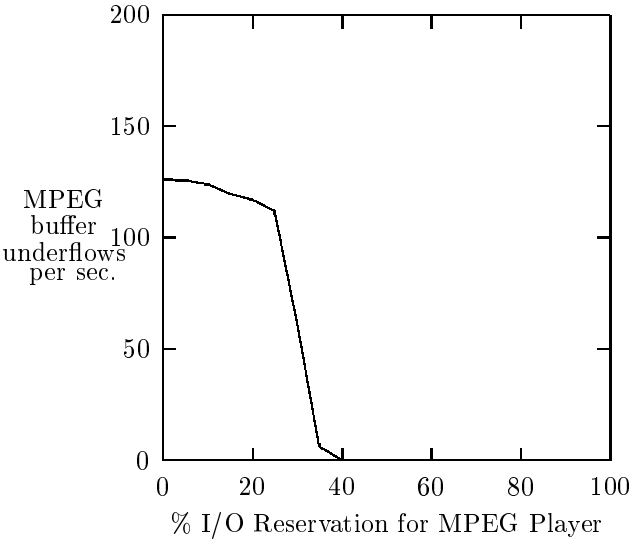

Figure 10: Buffer Underflows of an MPEG Player Under I/O Overload

The first experiment runs a MPEG player under CPU and I/O overload. The MPEG-1 player has 10 read-ahead threads, that read a movie from the local disk and send it to the MPEG decoder card. The MPEG player requires both I/O and CPU resources.

We repeatedly created CPU overload by running multiple copies of the `onoff` program. Each copy of this program consumes 11ms of CPU and then sleeps for 5ms. The `onoff` program poses a challenge to most CPU schedulers, since it consumes CPU with a duty cycle of about 60%, and it sleeps frequently.

7.2 Providing QoS in Overloaded System by Resource Reservation

Figures 9 and 10 depict the effects of resource reservation on an MPEG-1 player under CPU and I/O overloads, respectively. In the CPU load experiment, the MPEG player was run against 8 concurrent instances of the `onoff` program. In the I/O load experiment, the MPEG player was run against 10 I/O processes that were accessing random locations on the disk that holds the MPEG movie. In all cases, the file system was running in reservation domain #1 with a fixed 20% CPU reservation (where applicable). The MPEG player was running in reservation domain #2. The I/O workload was running in reservation domain #3, and each competing instance of `onoff` was running in its own reservation domain without any explicit reservations. In all cases, we varied the corresponding resource reservation of domain #2.

Figures 9 and 10 show the average number of buffer *underflows* in the MPEG card (which indicate missed deadlines). Buffer underflows are sampled at a 200Hz rate. A zero value indicates no missed deadlines, and it is always achieved when the MPEG player is running in an otherwise idle system. The value 200 indicates that all deadlines were missed (the buffer underflowed continuously).

This experiment demonstrates that *reservation domains provide guaranteed execution rate in an overloaded system.* Both MTR-LS and WRR provided good quality of service under CPU load. However, MTR-LS required a reservation of 15% to prevent underflows, while WRR required a reservation of 55%. The reason for MTR-LS superiority is that it prefers less greedy processes, such as the MPEG player, over more greedy processes, such as `onoff`, which exhausted its entire reservation. Note that each `onoff` consumes more than 60% of the CPU on an idle machine, and here it was reserved less than 9% of the CPU. The Plan9 priority scheduler failed in this experiment to prevent buffer underflows, since it did not prefer the MPEG player over the `onoff` processes. The Plan9 scheduler detects CPU bound processes, but it failed to recognize the `onoff` processes as CPU hogs since they sleep frequently.

7.3 Comparison of MTR-LS with Priority Scheduling and Weighted Round-Robin

Figure 11 compares the scheduling delays of MTR-LS with the Plan9 priority scheduler and WRR. We run a single copy of `sleeper` concurrently with a varying

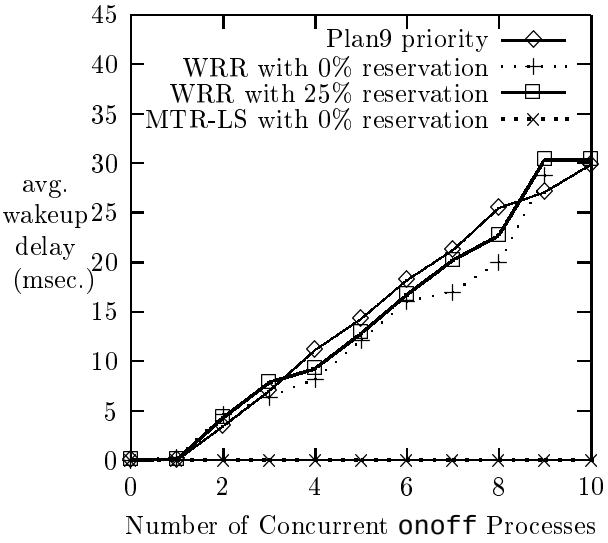

Figure 11: Wakeup Delay in Overloaded Systems

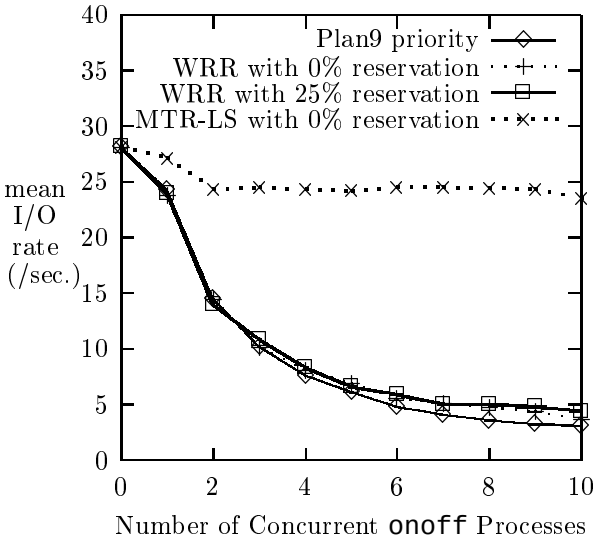

Figure 12: I/O Rate in Overloaded Systems

number of `onoff` programs under different scheduling algorithms. The `sleeper` program consumes 11msec. of CPU time and then sleeps for 100 msec. repeatedly. `Sleeper` measures the difference between its actual wakeup time and its anticipated wakeup time and reports the average delay. The `sleeper` program was run in reservation domain #2, and each instance of `onoff` was run in a separate reservation domain without any explicit CPU reservation.

Figure 11 illustrates the advantage of MTR-LS over Plan9 priority scheduling and WRR. MTR-LS always schedules the `sleeper` program before `onoff`, since `sleeper` consumes CPU at a slower rate, and thus its tokens tend to migrate to the beginning of the list $\mathcal{L}$. In other words, *MTR-LS prefers less-greedy processes*. In fact, this experiment shows that MTR-LS provides a delay bound to processes that do not exceed their reservation.

Of course, if `sleeper` had an explicit priority (`nice`) in the priority scheduler, it would not suffer from wakeup delays. However, changing the process priority may not be available in many situations, and may cause undesired side effects.

7.4 CPU Scheduling of I/O Bound Processes

Figure 12 depicts the I/O rate of a single thread of an I/O bound program when it runs concurrently with a varying number of `onoff` programs under different scheduling algorithms. The I/O bound program was run in reservation domain #2, and each instance of the `onoff` program was run in a separate domain without any explicit CPU reservation.

The most interesting result of this experiment is that the I/O rate *decreased* under increasing CPU load with the Plan9 priority scheduling and WRR. The explanation of this phenomenon is as follows: Although the I/O process has an unlimited access to the disk, it must wait for its turn for the CPU in order to issue the next I/O request. The I/O process is never granted the CPU immediately in WRR, since it is always appended to tail of the ready queue after the n `onoff` processes. This is why I/O throughput decreased as CPU contention increased. Providing CPU reservation to both the I/O process and to the file system process did not alleviate this problem. Plan9 priority scheduling failed to prefer the I/O process over the `onoff` processes, because all of them sleep frequently.

MTR-LS successfully isolated the I/O process from the CPU overload, even without any CPU reservations. The reason is that the I/O process is less CPU-greedy than the `onoff` processes. In other word, MTR-LS provided a *cumulative service* guarantee for the I/O bound process, which was not delayed by the CPU.

8 Conclusions and Future Work

In this paper we introduced a new operating system abstraction, called *reservation domains*, which can be used to provide predictable quality of service in overloaded systems and to partition resources among concurrent users. Reservation domains allow soft real-time appli-

cations to co-exist with batch applications in the same system, without any change to the applications.

We also described a new scheduling algorithm, Move-to-Rear List Scheduling (MTR-LS), that provides fairness, delay bound, and cumulative service guarantees. We explained why those QoS parameters are important. The rest of the paper described Eclipse, a new experimental operating system, that implements reservation domains and MTR-LS scheduling. Eclipse can schedule three types of resources: CPU, disk I/O and physical memory (working set). Experimental results indicate that Eclipse provide delay bounds and cumulative service bounds in overload situations. We also showed that MTR-LS prefers *less-greedy* processes, which means that it automatically provides better performance in many cases, as shown by our experiments. We compared MTR-LS with Plan9 priority scheduling and with weighted round-robin scheduling. MTR-LS provided superior results in several cases. We illustrated a simple implementation of MTR-LS in the Appendix. More time efficient implementations of MTR-LS are possible, but they use more complex data structures.

Our ongoing work concentrates on extending the reservation domain approach into a distributed environment with multiple clients and servers, investigating hierarchical scheduling for reservation domains, developing new scheduling algorithms, and porting the reservation domains and MTR-LS into more popular operating systems, such as Linux or FreeBSD.

Acknowledgments

The authors would like to thank Ron Phelps for helping with programming the experiments.

References

[1] J. Bruno, E. Gabber, B. Özden, H. Saran, and A. Silberschatz. Closed-loop packet sources and cumulative service. Technical report, Aug 1997.

[2] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing qos guarantees. In *Proceedings of ACM Multimedia, Seattle, Washington*, November 1997.

[3] A. Demers, S. Keshav, and S. Shenker. "Design and Analysis of a Fair Queuing Algorithm". In *Proceedings of the ACM SIGCOMM Austin, Texas, September 1989*, September 1989.

[4] P. Goyal, X. Guo, and H. M. Vin. "A Hierarchical CPU Scheduler for Multimedia Operating Systems". In *Proceedings of the USENIX 2nd Symposium on Operating System Design and Implementation Seattle, Washington, October 1996*, October 1996.

[5] M. B. Jones, D. Roşu, and M.-Cătălin Roşu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 198–211, Saint-Malo, France, October 5-8 1997.

[6] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Braham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, September 1996.

[7] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[8] J. Niehand and M. S. Lam. The design and evaluation of smart: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 184–197, Saint-Malo, France, October 5-8 1997.

[9] B. Özden, R. Rastogi, and A. Silberschatz. *Fellini Continuous Media Storage Server*. Kluwer Academic Publishers.

[10] *Plan 9 Programmer's Manual, Second Edition*, volume 1 and 2. Computing Science Research Center, AT&T Bell Labs, Murray Hill, NJ, 1995. ISBN 0-03-017138-5 and 0-03-017139-3.

[11] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks–the single node case. *IEEE/ACM Transactions on Networking*, pages 344–357, June 1993.

[12] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems, The Journal of the USENIX Association*, 8(3):221–254, Summer 1995.

[13] I. Stoica and et.al. "A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems". In *Proceedings of IEEE Real-Time Systems Symposium*, December 1996.

[14] M. Thorup. "On RAM Priority Queues". In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, 1996*, pages 59–67, Atlanta, January 1996.

[15] Ian Wakeman, Atanu Ghosh, and Jon Crowcroft. Implementing real time packet forwarding policies using streams. In *Proceedings of USENIX 1995 Technical Conference*, pages 71–82, New Orleans, Louisiana, January 16-20 1995.

[16] C. A. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report TM-528, MIT, Laboratory for Computer Science, June 1995.

[17] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First Symposium on Operating System Design and Implementation (OSDI)*, pages 1–11, Montrey, California, November 14-17 1994.

Appendix: The Eclipse Implementation of MTR-LS

```
typedef struct Proc  Proc;
typedef struct Rd   Rd;

enum {
  /* clock ticks per sched. cycle */
  CYCLE2TK = (HZ/2),
};

struct Proc {
  Proc *rnext;  /* next process in run q */
  int  state;
  Rd   *rd;     /* reservation domain */
};

struct Rd {
  Rd   *next;   /* next RD in list */
  int  rdid;    /* reservation domain ID */
         /* 0 indicates a kernel proc. */
  int  cpu_resrv; /* CPU reservation in */
         /* ticks */
  uint lo_tok;  /* lowest token time-stamp */
  /* cyclic list of time-stamps */
  uint tok_list[CYCLE2TK];
  int  head, tail;
};

/* Globals */
static Schedq runq; /* process run queue */
static RDq rd;      /* reservation domains */
uint hi_stamp;      /* current time-stamp */

/* Initialization of tokens list */
void
init_tok_list(void)
{
  Rd *r;
  int go;

  /* first token in each domain */
  hi_stamp = 0;
  for (r = rd.head; r; r = r->next) {
    r->lo_tok = r->tok_list[0] = hi_stamp++;
    r->head = r->tail = 1;
  }

  /* assign tokens in interleaved fashion */
  do {
    go = 0;
    for (r = rd.head; r; r = r->next)
      if (r->head < r->cpu_resrv) {
        go = 1;
        r->tok_list[r->head++] = hi_stamp++;
      }
  } while(go);
}

/* Clock interrupt routine */
static void
clock(Ureg *ur, void *arg)
{
  Proc *p;
  Rd *r;

  p = m->proc;

  if (p) {
    if (p->state == Running) {
      r = p->rd;
      if (++hi_stamp == 0)
        /* re-initialize tokens list at */
        /* time-stamp overflow */
        init_tok_list();
      r->tok_list[r->head++] = hi_stamp;
      if (r->head >= CYCLE2TK)
        r->head = 0;
      r->lo_tok = r->tok_list[r->tail++];
      if (r->tail >= CYCLE2TK)
        r->tail = 0;
    }
  }

  if (u && p && p->state == Running) {
    /* preempt if not holding a spin lock */
    ...
    sched();
  }
}

/* Select the next process to run */
Proc*
runproc(void)
{
  Proc *p, *prev, *best, *prevb;

loop:
  prev = best = 0;
  for (p = runq.head; p; p = p->rnext) {
    /* run kernel processes first */
    if (p->rd->rdid == 0) {
      best = p;
      prevb = prev;
      break;
    }

    if (best == 0 ||
        p->rd->lo_tok < best->rd->lo_tok) {
      best = p;
      prevb = prev;
    }
    prev = p;
  }

  if (best == 0)
    goto loop;

  /* remove process from run queue */
  remq(&runq, best, prevb);
  return best;
}
```