Improving Linux resource control using CKRM

Shailabh Nagar
IBM T.J. Watson Research Center
nagar@watson.ibm.com

Rik van Riel
Red Hat, Inc.
riel@redhat.com

Hubertus Franke
IBM T.J. Watson Research Center
frankeh@watson.ibm.com

Chandra Seetharaman
IBM Linux Technology Center
chandra.sekharan@us.ibm.com

Vivek Kashyap
IBM Linux Technology Center
vivk@us.ibm.com

Haoqiang Zheng
Columbia University
hzheng@cs.columbia.edu

Abstract

One of the next challenges faced in Linux kernel development is providing support for workload management. Workloads with diverse and dynamically changing resource demands are being consolidated on larger symmetric multiprocessors. At the same time, it is desirable to reduce the complexity and manual involvement in workload management. We argue that the goal-oriented workload managers that can satisfy these conflicting objectives require the Linux kernel to provide class-based differentiated service for all the resources that it manages. We discuss an extensible framework for class-based kernel resource management (CKRM) that provides policy-driven classification and differentiated service of CPU, memory, I/O and network bandwidth. The paper describes the design and implementation of the framework in the Linux 2.6 kernel. It shows how CKRM is useful in various scenarios including the desktop. It also presents preliminary performance evaluation results that demonstrate the viability of the approach.

1 Introduction

Workload management is an increasingly important requirement of modern enterprise computing systems. There are two trends driving the development of enterprise workload management middleware. One is the consolidation of multiple workloads onto large symmetric multiprocessors (SMPs) and mainframes. Their diverse and dynamic resource demands require workload managers (WLMs) to provide efficient differentiated service at finer time scales to maintain high utilization of expensive hardware. The second trend is the move towards specification of workload performance in terms of the business importance of the workload rather than in terms of low-level system resource usage. This has led to the increasing use of goal-oriented workload managers, described shortly, which are more tightly integrated into the business processes of an enterprise.

Traditional system administration tools have been built with two layers. The lower, OS specific layer deals with modifying and monitoring operating system parameters. The upper

layer(s) provide an OS independent API, generally through a graphical user interface, allowing a multi-tier or clustered system to be managed through a unified API despite containing heterogenous operating systems. While such tools provide a convenient administrative interface to heterogeneous operating systems they do little to address the complexity of managing workloads that span multiple tiers. The burden of translating business goals into workload resource requirements and the latter into OS specific tuning parameters remains on the system administrators. Increasing workload consolidation only adds more complexity to an already onerous problem.

As described in [7], the first stage in improving workload management are *entitlement-based* workload managers (WLMs) such as [9, 5, 11] which enforce entitlements or shares on resources consumed by groups of processes, users, etc. This allows the more important groupings to see improved response times and higher bandwidth due to preferential access to the server hardware. As importantly, it allows expensive SMP servers to have higher utilizations since system administrators can afford to load them more without fear of penalizing the important groupings.

However, the complexity of determining the right entitlements (henceforth called shares) for a grouping remains on the human system administrator. Not only does s/he need to map the importance of a workload to its entitlements, s/he also needs to adjust these shares dynamically when the demand and/or importance of *any* workload changes. Such dynamic share changes have become increasingly difficult to compute in a timely manner when manual involvement is part of the adaptive feedback loop.

To address the complexity of share specifications, *goal-oriented workload managers* have been developed [1, 10] which allow a system to be more self-managed. Such WLMs allow the human system administrator to specify high level performance objectives in the form of policies, closely aligned with the business importance of the workload. The WLM middleware then uses adaptive feedback control over OS tuning parameters to realize the given objectives.

In mainstream operating systems, including Linux, the control of key resources such as memory, CPU time, disk I/O bandwidth and network bandwidth is typically strongly tied to processes, tasks and address spaces and are highly tuned to maximize system utilization. This introduces additional complexity to the WLM which needs to translate the QoS requirements into these low level per task requirements, tough typically QoS is enforced at work class level. Hence, in order to isolate the autonomic goal oriented layers of the system management from the intricacies of the operating system, we introduce the class concept into the operating system kernel and require the OS to provide differentiated service for all major resources at a class granularity defined by the WLM.

In this paper, we discuss a framework called class-based kernel resource management (CKRM) that implements this support under Linux. In CKRM, a class is defined as a dynamic grouping of OS objects of a particular type (classtype) and defined through policies provided by the WLM. Each class has an associated share of each of its resources. For instance, CKRM tasks classes provides resource management for four principal physical resources managed by the kernel namely CPU time, physical memory pages, disk I/O and bandwidth. Sockets classes provide inbound network bandwidth resource control. The Linux resource schedulers are modified to provide differentiated service at a class granu-

larity based on the assigned shares. The WLM can dynamically modify the composition of a class and its share in order to meet higher level business goals. We evaluate the performance of the CKRM using simple benchmarks that demonstrate the efficacy of its approach.

This work makes several contributions that distinguish it from previous related work such as resource containers [2] and cluster reserves [4]. First, it describes the design of a flexible kernel framework for class-based management that can be used to manage both physical and virtual resources (such as number of open files). The framework allows the various resource schedulers and classification engine to be developed and deployed independent of each other. Second, it shows how incremental modifications to existing Linux resource schedulers can make them provide differentiated service effectively at a class granularity. To our knowledge, this is the first open-source resource management package that attempts to provide control over all the major physical resources—i.e., CPU, memory, I/O, and network. Third, it provides a policy-driven classification engine that eases the development of new higher level WLMs and enables better coordination between multiple WLMs through policy exchange. Thirdly, through the resource class filesystem the WLM goals can be manipulated by normal users, making it useful on the desktop. Finally, it develops a tagging mechanism that allows server applications to participate in their resource management in conjunction with the WLM.

The rest of the paper is organized as follows. Section 2 gives an overview of CKRM and its core bits. Sections 3 briefly describes the classification engine. Section 4 presents the facilities provided by CKRM for monitoring. The inbound network controller, the first major controller ported to CKRM's new interface, is described in Section 5. Section 6 describes the filesystem interface which replaces the system call interface used in CKRM's earlier design presented in OLS 2003 [13]. Section 7 describes how CKRM might be used, both on a desktop system and on some server workloads. Section 8 concludes with directions for future work in the project.

2 Framework

A typical WLM defines a workload to be any system work with a distinct business goal. From a Linux operating system's viewpoint, a workload is a set of kernel tasks executing over some duration. Some of these tasks are dedicated to this workload. Other tasks, running server applications such as database or web servers, perform work for multiple workloads. Such tasks can be viewed as executing in phases with each phase dedicated to one workload. Server tasks can explicitly inform the WLM of its phase by setting an application tag. A WLM can also infer the phase by monitoring significant system events such as forks, execs, setuid, etc. and classifying the server task as best as possible.

In this scenario, a WLM translates a high level business goal of a workload (say response time) into system goals for the set of tasks executing the workload. The system goals are a set of delays seen by the workload in waiting for individual resources such as CPU ticks, memory pages, etc. The WLM monitors the business goals, possibly using application assistance, and the system usage of its resources. If the business goal is not being met, it identifies the system resource(s) which form a performance bottleneck for the workload and adjusts the workload's share of the resource appropriately. The CKRM framework enables a WLM to regulate workloads through a number of components, as shown in Fig. 1:

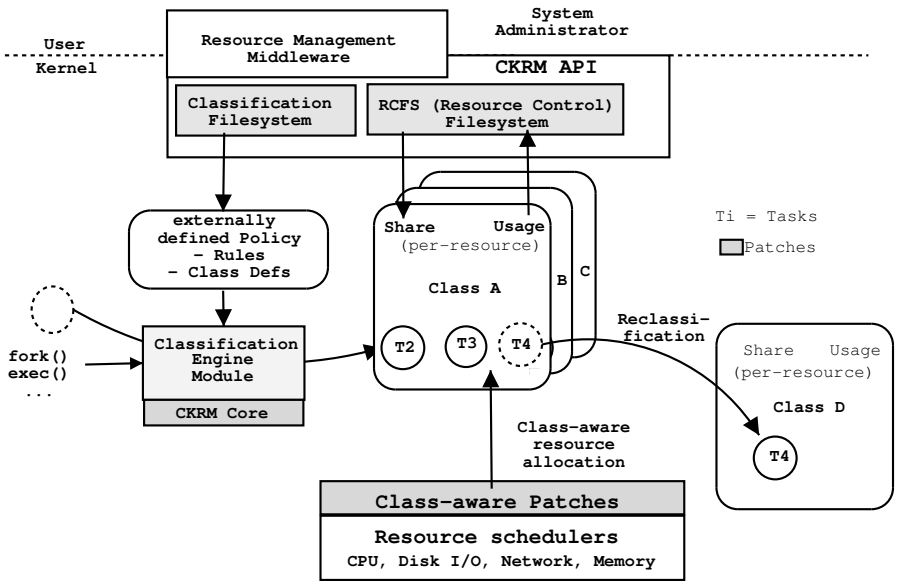

Figure 1: CKRM lifecycle

Core: The core defines the basic entities used by CKRM and serves as the link between all the other components. A class is a group of kernel objects with an associated set of constraints for resource controllers operating on those kernel objects—e.g., a class could consist of a group of tasks which have a joint share of cpu time and resident page frames. Each class has an associated classtype which identifies the kernel object being grouped. CKRM currently defines two classtypes called `task_class` and `socket_class` for grouping tasks and sockets. For brevity, the term taskclass and socketclass will be used to denote a class of classytpe `task_class` and `socket_class` respectively. Classtypes can be enabled selectively and independent of each other. A user not interested in network regulation could choose to disable `socket_classes`. Classes in CKRM are hierarchical. Children classes can be defined to subdivide the resources allocated to the parent.

Classification engine (CE): This optional components assists in the association of kernel objects to classes of its associated classtype. Each kernel object managed by CKRM is always associated with some class. If no classes are defined by the user, all objects belong to the default class for the classtype. At significant kernel events such as fork, exec, setuid, listen, when the attributes of a kernel object are changed, the Core queries the CE, if one is present, to get the class into which the object should be placed. CE's are free to use any logic to return the classification. CKRM provides a rule-based classification engine (RBCE) which allows privileged users to define rules which use attribute matching to return the class. RBCE is expected to meet the needs of most users though they can define their own CE's or choose not to have any and rely upon manual classification of each kernel object through CKRM's rcfs user interface (described later).

Resource Controllers: Each classtype has a set of associated resource controllers, typically one for each resource associated with the classtype—e.g., taskclasses have cpu, memory, and I/O controllers to regulate the cpu ticks, resident page frames and per-disk I/O bandwidth consumed by it while socketclasses have an accept queue controller to regulate the num-

ber of TCP connections accepted by member sockets. Resource requests by a kernel object in a class are regulated by the corresponding resource controller, if one exists and is enabled. The resource controllers are deployed independent of each other so a user interested only in controlling CPU time for taskclasses could choose to disable the memory and I/O controllers (as well as the socketclass classtype and all its resource controllers).

Resource Control File System (RCFS): It forms the main user-kernel interface for CKRM. Once RCFS is mounted, it provides a hierarchy of directories and files which can be manipulated using well-known file operations such as open, close, read, write, mkdir, rmdir and unlink. Directories of rcfs correspond to classes. User-kernel communication of commands and responses is done through reads/writes to virtual files in the directories. Writes to the virtual files trigger CKRM Core functions and responses are available through reads of the same virtual file.

The CKRM architecture outlined above achieves three major objectives:

- Efficient, class-based differentiation of resource allocation and monitoring for dynamic workloads: Regulate and monitor kernel resource allocation by classes which are defined by the privileged user and not only in terms of tasks. The differentiation should work in the face of relatively rapid changes in class membership and over roughly the same time intervals at which process-centric regulation currently works.

- Low overhead for non-users: Users disinterested in CKRM's functionality should see minimum overhead even if CKRM support is compiled into the kernel. Signs of user disinterest include omitting to mount rcfs or not defining any classes. Even for users, CKRM tries to keep overheads proportional to the features used.

- Flexibility and extensibility through minimization of cross-component dependencies: Classification engines should be independent of classtypes and optional, classtypes should be independent of each other and so should resource controllers, even within the same classtype. This goal is achieved through object-oriented interfaces between components. Minimizing dependencies allows kernel developers to selectively include components based on their perception of its utility, performance and stability. It also permits alternative versions of the components to be used depending on the target environment—e.g., embedded Linux distributions could have a different set of taskclass resource controllers (or even classtypes) than server-oriented distributions.

3 Classification

The Classification Engine (CE) is an optional component that enables CKRM to automatically classify kernel objects within the context of its classtype. Since the CE is optional and since we want to main flexibility in its implementation, functionality and deployment, it is supplied as a dynamically loadable module. The CE interacts with CKRM core as follows. The CKRM core defines a set of ckrm events that constitute a point during execution where a kernel object could potentially change its class. A classtype can register a callback at any of these events. As an example, the task class hooks the fork, exec, exit, setuid, setgid calls where as the socket class hooks the listen and accept calls. In these callbacks the classtypes typically invoke the optional CE to obtain a new class. If no CE is registered or the

CE does not determine a class, the object remains in its current class, otherwise the object is moved to the new class and the corresponding resource managers of that class's type are informed about the switch.

For every classtype the CE wants to provide automatic classification for, it registers a classification callback with the classtype and the set of events to which the callback is limited to. The task of CE is then to provide a target class for the kernel objects passed in the context of the classtype. For instance, task classes pass only the task, while socket classes pass the socket kernel object as well as the task object. Though the implementation of the classification engine is completely independent of CKRM, the CKRM project provides a default classification, called RBCE, that is based on classification rules. Rules consist of a set of rule terms and a target class. A rule term specifies one particular kernel object attribute, a comparision operator (=,<,>,!) and a value expression. To speed up the classification process we maintain state with tasks about which rules and rule terms have been examined for a particular task and only reexamine those terms that are indicated by the event. RBCE provides rules based on task parameters ((pid, gid, uid, executable) and socket information (IP info). The rules in conjunction with the defined classes constitute a site policy for workload managment and is dynamically changable (See user interface section) into the RBCE. Hence, this approach ensures the separation of policy and enforcement.

To facilitate the interaction with WLMs to provide event monitoring and tracing, the CE can also register a notification callback with any classtype, that is called when a kernel object is assigned to a new class. Similar so the classification callback, the notification callback can be limited to a set of ckrm events. This facility is utilized in resource monitoring, described next.

4 Monitoring

We now describe the monitoring infrastructure. Strictly speaking, the per-class monitoring components are part of CKRM while the per-process components are not. However, we shall describe them together as they both can be utilized by goal-based WLMs. Furthermore, they are bundled with the classification engine and utilize the CE's notification callback to obtain classification events. The monitoring infrastructure illustrated in Fig. 2 is based on the following design principles:

1. Event-driven: Every significant event in the kernel that affectsthe state of a task is recorded and reported back to the state-agent. The events of importance are aperiodic such as process fork, exit and reclassification as well as periodic events such as sampling. Commands sent by the state-agent are also treated as events by the kernel module.

2. Communication Channel: A single logical communication channel is maintained between the state-agent and the kernel module and is used for transferring all commands and data. Most of the data flow is from the kernel to user space in the form of records resulting from events.

3. Minimal Kernel State: The design minimizes the additional per-process state that needs to be maintained within the kernel. Most of the state needed for high level control purposes is kept within the state agent and updated through the records sent by the kernel.

The state-agent, which can also be integrated within a WLM, maintains state on each existing and exited task in the system and provides it to the WLM. Since the operating system

does not retain the state of exited processes, the stateagent must maintain it for future consumption by the WLM. The state-agent communicates with a kernel module through a single bidirectional communication chan-nel, receiving updates to the process state in the form of records and occasionally sending com-mands. Events in the kernel such as process fork, exit, reclassify (resulting from change in any process attribute such as gid, pid) cause records to be generated through functions provided by the kernel module.

Server tasks can assist the WLM by informing it about the phase in which they are operating (each phase corresponds to a workload). Such tasks invoke CKRM to set a tag associated with their `task_struct` in the kernel. CKRM uses this event to reclassify the task and also records the event (to be transmitted to the WLM through the state-agent). Other kernel events that might cause a task to be reclassified (such as the exec and setuid system calls, etc.) are also noted by CKRM and passed to the WLM through the state-agent. In addition, CKRM performs periodic sampling of each task's state in the kernel to determine the resource it is waiting on (if any), its resource consumption so far and the class to which it belongs. The sample information is transmitted to the state-agent. The WLM can correlate the information with the tag setting to statistically determine the resource consumption and delays of both server and dedicated processes executing a workload. Sampling is done through a kernel module function that is invoked by a selfrestarting kernel timer. Commands sent by the state-agent cause appropriate functions in the kernel module to execute and also return data in the form of records. The kernel components are kept simple and only minimal additional state has to be maintained in the kernel. In particular, the kernel does not have to maintain extra state about exited processes which introduces problems with PID reusage, memory management to name a few. Instead, relevant task information is replicated in user space, is by definition received in the correct time order (see below) and can be kept around until the WLM has consumed the information. Furthermore, the semantics of a reclassification in the kernel, which identifies a new phase in a server process, does not have to be introduced into the kernel space.

The following small changes are required to the linux kernel to track system delays. The `struct delay_info` is added to the `task_struct`. Delay_info contains 32-bit variables to store cpu delay, cpu using, io delay and memory io delay. The counters provide micro second accuracy. The current cpu scheduler records timestamps whenever i) a task becomes runnable and is entered into a runqueue and ii) when a context switch occurs from one task to another. We use these same timestamps to get per-task cpu wait and cpu using times recorded respectively. I/O delays are measured by the difference of timestamps taken when a task blocks waiting for I/O to complete and when it returns. All I/O is normally attributed to the blocking task. Page-fault delays, however, are treated as special I/O delays. On entrance to and exit from the page fault handler the task is marked or unmarked as being in a memory path using flags in `task_struct`. If during the I/O delay, this flag is set, the I/O delay is counted as a memory delay instead of as a pure I/O delay. The per-task delay information is accessible through the file `/proc/<pid>/delay`. Similarly, each class contains a `delay_info` structure.

In contrast to the precise accounting of delays, sampling examines the state of tasks at fixed interval. In particular, we sample at fixed intervals (~1sec) the entire set of tasks in the system and increment per task counters that are integrated into the task private structure attached

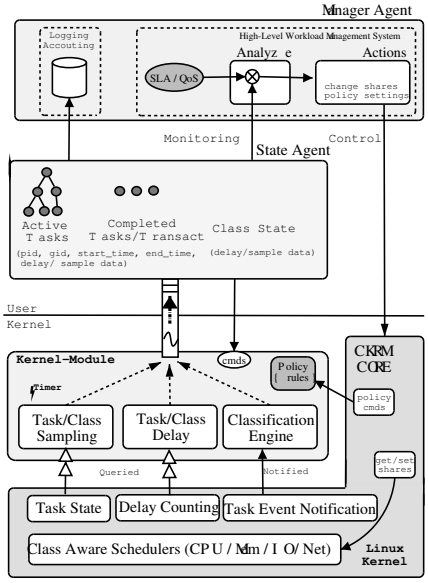

Figure 2: CKRM lifecycle

by the classification engine that builds the core of the kernel module. We increment counters if a task is running, waiting to run, performing I/O or handles a pagefault I/O. Task data (sampled and/or precise) is requested by and sent to the state-agent in coarser intervals. We can send data in continuous aggregate mode or in delta mode, i.e. only if task data has changed do we send a new data record and then reset the local counters. The task transition events are sent at the time they occur. We distinguish the fork, exit, and reclassification events as records. At each reclassification (which could potentially be the end of a phase) we transmit the sample and delay data and reset them locally.

As a communication channel we utilize the linux relayfs pseudo filesystem, a highly efficient mechanism to share data between kernel and user space. The user accesses the shared buffers, called channels, as files, while the kernel writes to them using buffer reservations and memory read/write operations. The content and structure of the buffer is determined by the kernel and user client. Currently the communication channel is self pacing. The underlying relayfs channel buffer will dynamically resize upto a maximum size. If for any reason the relayfs buffer overflows, record sending will automatically stop, an indication is sent and the state-agent will have to drain the channel and request a full state dump from the kernel.

We have measured the data rate during a standard kernel build, which creates a significant amount of task events (fork,exec,exits). For a 2-CPU system with 2 seconds sample collection we observed a data rate of 8KB/second and a total of 190 records/sec, well within a limit that can be processed without creating significant overhead in the system.

5 Inbound Network

Various OS implementations offer well established QoS infrastructure for outbound bandwidth management, policy-based routing and Diffserv [3]. Linux in particular, has an elaborate infrastructure for traffic control [8] that consists of queuing disciplines(qdisc) and filters. A qdisc consists of one or more queues and a packet scheduler. It makes traffic con-

form to a certain profile by shaping or policing. A hierarchy of qdiscs can be constructed jointly with a class hierarchy to make different traffic classes governed by proper traffic profiles. Traffic can be attributed to different classes by the filters that match the packet header fields. The filter matching can be stopped to police traffic above a certain rate limit. A wide range of qdiscs ranging from a simple FIFO to classful CBQ or HTB are provided for outbound bandwidth management, while only one ingress qdisc is provided for inbound traffic filtering and policing. The traffic control mechanims can be used invarious places where bandwidth is the primary resource to control.

Due to the above features, Linux is widely used for routers, gateways, edge servers; in other words, in situtations where network bandwidth is the primary resource to differentiate among classes. When it comes to endservers networking, QoS has not received as much attention since QoS is primarily governed by the systems resources such as memory, CPU and I/O and less by network bandwidth. When we consider end-to-end service quality, we should require networking QoS in the end servers as exemplified in the fair share admission control mechanism proposed in this section.

We present a simple change to the existing TCP accept mechanism to provide differentiated service across priority classes. Recent work in this area has introduced the concept of prioritized accept queues [6] and accept queue schedulers using adaptive proportional shares to self-managed web [14]. In a typical TCP connection, the client initiates a request to connect to a server. This connection request is queued in a global accept queue belonging to the socket associated with the server's port. The server process picks up the next queued connection request and services it. In effect, the incoming connections to a particular TCP socket are serialized and handled in FIFO order. When the incoming connection request load is higher than the level that can be handled by the server requests have to wait in the accept queue until the next can be picked up.

We replace the existing single accept queue per socket with multiple accept queues, one for each priority class. Incoming traffic is mapped into one of the priority classes and queued on the accept queue for that priority. The accept queue implements a weighted fair scheduler such that the rate of acceptance from a particular accept queue is proportional to the weight of the queue. In the first version of the priority accept queue design initially proposed by the CKRM project [13], starvation of certain priority classes was a possibility as the accepting process picked up connection requests in the order of descending priority.

The efficacy of the proportional accept queue mechanism is demonstrated by an experiment. We used Netfilter [12] to MARK options to characterize traffic into two priority classes with respective weights of 3:1. The server process utilises a configurable number of threads to service the requests. The results are shown in Figure 3. When the load is low and there are service threads available no differentiation takes place and all requests are processed as they arrive. Under higher load, requests are queued in the accept queue with class 1 receiving a proportionally higher service rate than class 2. The expriment was repeated, maintaining a constant inbound connection request rate. The proportions of the two classes were then switched to see the service rate for the two classes reverse as seen in Figure 4

6 Resource Control Filesystem

In the Linux kernel development community, filesystems have become very popular as user

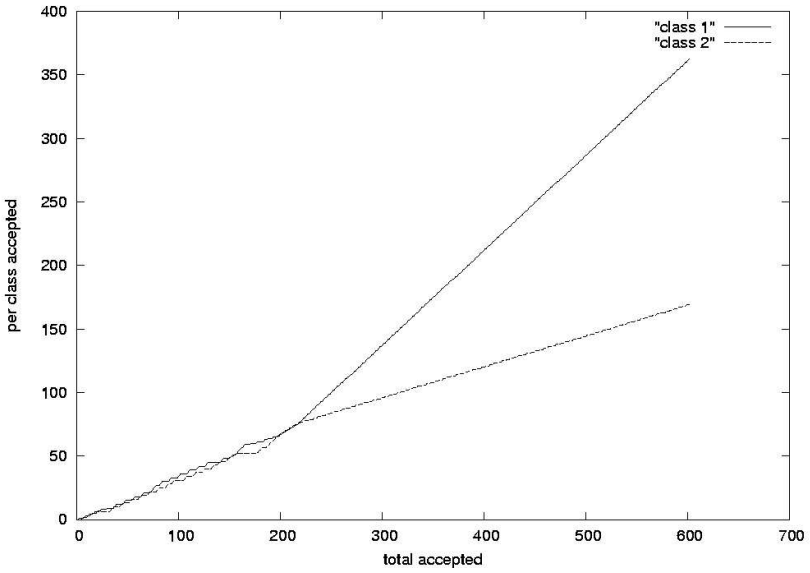

Figure 3: Proportional Accept Queue: Results

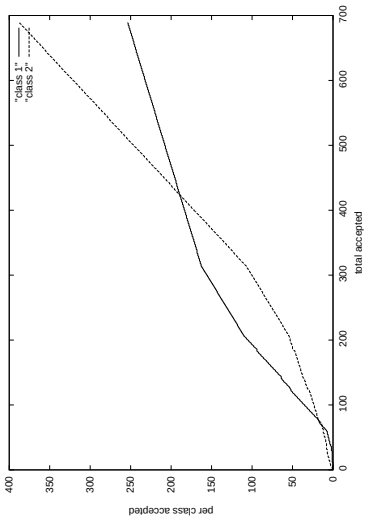

Figure 4: Proportional Accept Queue: Results under change

interfaces to kernel functionality, going well beyond the traditional use for disk-based persistent storage. The Linux kernel's object-oriented Virtual File System (VFS) makes it easy to implement a custom filesystem. Common file operations like open, close, read and write map naturally to initalization, shutdown, kernel-to-user and user-to-kernel communication. For CKRM, the tree structured namespace of a filesystem offers the additional benefit of an intuitive representation of the class hierarchy. Hence CKRM uses the Resource Control Filesystem (RCFS) as its user interface.

The first-level directories in RCFS contain the roots of subtrees associated with classtypes build or loaded into the kernel (`socket_class` and taskclass currently) and the clas-

sification engine (ce). Within the classtype subtrees, directories represent classes. Users can create new classes by creating a directory as long as they have the proper access rights. Within the `task_class` directory, each directory represents a task class. `/rcfs/taskclass`, the root of the `task_class` classtype, represents the default taskclass which is always present when CKRM is enabled in the kernel. Each `task_class` directory contains a set of virtual files that are created automatically when the directory is created. Each virtual file has a specific function as follows:

1. members: Reading it gives the names of the tasks in the taskclass.
2. config: To get/set any configuration parameters specific to the taskclass.
3. target: Writing a task's pid to this file causes the task to be moved to the taskclass, overriding any automatic classification that may have been done by a classification engine.
4. shares: Writing to this file sets new lower and upper bounds of the resource shares for the taskclass for each resource controller. Reading the file returns the current shares. The controller name is specified on a write which makes it possible to set the values for controllers independent of each other.
5. stats: Reading the file returns the statistics maintained for the taskclass by each resource controller in the system. Writing to the file (specifying the controller) resets the stats for that controller.

The `socket_class` directory is somewhat similar. Directories under `/rcfs/socket_class/` represent listen classes and have the same magic files as task_classes. Whereas task_classes use the pid to identify the class member, socket_classes, which group listening sockets, use ip address + port name to identify their members. Within each listen class, there are automatically created directories, one for each accept queue class. The accept queue directories, numbered 1 through 7, have their own shares and stats virtual files similar to those for `task_classes`.

The `/rcfs/ce` directory is the user interface to the optional classification engine. It contains the following virtual files and directory:

1. reclassify: writing a pid or ipadress+port to the file causes the corresponding task or listen socket to be put back under the control of the classification engine. On subsequent significant kernel events, the ce will attempt to reclassify the task/socket to a new taskclass/socketclass if the task/sockets attributes have changed.
2. state: to set/get the state (active or inactive) of the classification engine. To allow a new policy to be loaded atomically, CE's can be set to inactive before loading a set of rules and activated thereafter.
3. Rules: The directory allows privileged users to create files with each file representing one rule. Reading the files, permitted for all, gives the classiication policy which is currently active. The ordering of rules in a policy is determined either by creation time of the corresponding file or by an explicitly specified order number within the file. The rule files contain rule terms consisting of attribute-value pairs and a target class. E.g., the rule *gid=10, cmd = bash, target = /rcfs/taskclass/A* indicates that tasks with gid 10 and running the bash program (shell) should get reclassified to task_class A.

7 Example uses

In this section we will describe a number of uses for CKRM, ranging from the traditional large server workload consolidation, to a university shell server, to the desktop—a novel use of workload management systems, made possible through the resource class filesystem.

7.1 Workload Consolidation

The classical use of a workload management system is workload consolidation, whether it's multiple departmental database servers on one large server, or one small server balancing resources between apache, ftpd, postfix and the interactive users. In either scenario the main objective is to make sure that none of the workloads can, through excessive resource use, cause the machine to become unusable for any of the others.

The simple solution is to start each of the services up in their own resource class and guaranteeing a certain amount of resources (say, 10% of the CPU and 20% of memory) for each of the services. Simultaneously the services can also have resource limits (say, 50% of memory). This combination of guarantees and limits gives the system a certain amount of freedom to balance the actual amount of resources each workload gets, while still putting effective guarantees and limits in place.

7.2 Shell Server

A shell server at a university faces a number of challenges. For example, the staff and postdocs should be protected from the load the students put on the machine and the students should be protected from each other. Similarly, batch jobs will usually have larger resource use limits (e.g. max cpu time used, max memory allocated), but a lower resource priority, as compared to any of the interactive programs. These problems can be solved by starting each class of process in the right process class.

On the other hand, if a staff member sends email to a student, the resources used by the student's mail filter should be accounted against that student's limits. This problem cannot be solved by having programs start out in a certain resource class, since the MTA process needs to transition between resource classes automatically. This can be solved by setting up a classification engine to automatically transfer a process to the *email* resource class when it execs `/usr/sbin/sendmail`. Similarly, when `/usr/bin/procmail` is being execed with a certain UID, the classification engine can move the process to the resource class where that user's interactive processes would normally run.

7.3 Desktop

With the right file and directory ownerships in the resource class filesystem, CKRM can be used in an area where traditional resource management systems tend to be cumbersome: on the desktop. A typical desktop configuration would have as its main goals that the system remains responsive to the user, no matter the background load, and would look something like the following.

The X server would get a good resource guarantee, e.g. 20% of CPU time and 20% of RAM. This makes sure that no matter what other processes run on the system, X can run smoothly and react to the console user with acceptably low latency.

At login time a PAM module would make sure that the rest of the user's processes get a good resource guarantee, too. An acceptable guarantee would be 50% of CPU time and 50% of RAM. This leaves enough resources free so that other things in the system can run (e.g. dis-

tro updates, updatedb, mail delivery), yet keeps most of the system dedicated to the user. The resource class created for the console user, e.g. `/rcfs/taskclass/console`, is set up to be writable for the console user. This way the user's processes can set resource guarantees and limits to certain classes of applications.

The user's GUI menu would take care of this subdividing of the resources guaranteed to the user. For example, the web browser could be restricted to 40% of RAM, so as to not put much pressure on the user's other processes. Multimedia processes could get part of the user's resource guarantees, e.g. 30% of the CPU and 10% of RAM guaranteed for the multimedia applications. This way the playback of multimedia should remain smooth, regardless of what the user's web browser and office suite are doing.

No superuser privileges are needed to configure these resource classes, or to move the user's processes between them. Any GUI framework or individual application will be able to determine the resources allocated to it, leading to more flexibility than possible with resource management systems that can only be configured by the super user. Note that since the user cannot raise the resource limits or guaranteed allocated to his main class, there should be no security risks involved with letting the user processes manipulate their own resource guarantees and limits.

8 Conclusion and Future Work

The consolidation of increasingly dynamic workloads on large server platforms has considerably increased the complexity of systems management. To address this, goal-oriented workload managers are being proposed which seek to automate low-level system administration requiring human intervention only for defining high level policies that reflect business goals.

In an earlier paper [13], we had argued that goal-oriented WLMs require support from the operating system kernel for class-based differentiated service where a class is a dynamic policy-driven grouping of OS processes. We had introduced a framework, called class-based kernel resource management, for classifying tasks and incoming network packets into classes, monitoring their usage of physical resources and controlling the allocation of these resources by the kernel schedulers based on the shares assigned to each class.

In this paper, we have described more details of the evolving design. In particular, CKRM has become more generic and supports groups of any kernel object involved in resource management, not just tasks. It has a new filesystem-based user API. Finally, the design introduces hierarchies into classes which permits greater flexibility for resource managers but also introduces challenges for CKRM controllers. A working prototype which includes an inbound network controller has been developed and made available through [15].

Future work in the project will involve redeveloping controllers for CPU, memory and I/O that are not only class-aware but can handle hierarchies of classes while keeping overheads low. Another important direction is the interactions of the resource schedulers and the impact of these interactions on the shares specified.

References

[1] J. Aman, C.K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger. Adaptive algorithms for Managing a Distributed Data Processing Workload. In *IBM Systems Journal*, volume 36(2), 1997.

[2] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, Dec 1998.

[4] J. Blanquer, J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. Resource management for qos in eclipse/bsd. In *Proc. FreeBSD 1999 Conference*, Oct 1999.

[5] IBM Corp. AIX 5L Workload Manager. http://www.redbooks.ibm.com/redbooks/SG245977.html.

[6] IBM DeveloperWorks. Inbound connection control home page. http://www-124.ibm.com/pub/qos/paq_index.html.

[7] Inc. D.H.Brown Associates. HP Raises the Bar for UNIX Workload Management. http://h30081.www3.hp.com/products/wlm/docs/hp.raises.bar.wkld.mgmt.pdf.

[8] Bert Hubert. Linux Advanced Routing & Traffic Control. http://www.lartc.org.

[9] Hewlett Packard Inc. HP Process Resource Manager. http://h30081.www3.hp.com/products/prm/.

[10] Hewlett Packard Inc. HP-UX Workload Manager. http://h30081.www3.hp.com/products/wlm/.

[11] Sun Microsystems Inc. Solaris Resource Manager. http://www.sun.com/software/resourcemgr/wp-srm/.

[12] J. Kadlecsik, H. Welte, J. Morris, M. Boucher, and R. Russel. Netfilter: Firewalling, NAT, and packet mangling for Linux 2.4. http://www.netfilter.org.

[13] S. Nagar, H. Franke, J. Choi, M. Kravetz, C. Seetharaman, V. Kashyap, and N. Singhvi. Class-based prioritized resource control in Linux. In *Proc. 2003 Ottawa Linux Symposium, Ottawa*, July 2003. http://ckrm.sf.net/documentation/ckrm-ols03-paper.pdf.

[14] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy. An Observation-based Approach Towards Self-Managing Web Servers. In *IWQoS 2002*, 2002.

[15] CKRM Open Source Project. Class-based kernel resource management. http://ckrm.sf.net/.

Proceedings of the Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*