Enabling Autonomic Workload Management in Linux

Hubertus Franke, Shailabh Nagar, Chandra Seetharaman, Vivek Kashyap
IBM Corp.
{frankeh,nagar,chandra.seetharaman,kashyapv}@us.ibm.com

Haoqiang Zheng
Columbia University
hzheng@cs.columbia.edu

Jiantao Kong
Georgia Tech University
jiantao@cc.gatech.edu

Abstract

Complexity reduction in workload management is driving the development of goal-oriented workload managers (WLMs). Simultaneously, server consolidation of workloads with dynamically changing resource demands calls for these WLMs to be increasingly efficient in managing resources. We propose the Class-based Kernel Resource Management (CKRM) framework, implemented in Linux, for operating systems to these requirements.

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 (WLM) middleware. One is allowing the human system administrator to only specify the business importance of a workload and let the WLM determine and enforce the workload's low-level system resource usage targets. This has led to the development of *goal-oriented* workload managers which are more tightly integrated into the business processes of an enterprise. The second trend is the consolidation of multiple workloads onto large symmetric multiprocessors (SMPs) and mainframes. Their diverse and dynamic resource demands require workload managers to provide efficient differentiated service at a fine time granularity to maintain high utilization of expensive hardware.

To be effective, a goal-oriented WLM requires the operating system kernel to provide *differentiated* service for *all* major resources at a granularity defined *by the WLM*. We propose a framework called class-based kernel resource management (CKRM) framework including class-aware CPU, memory, I/O and inbound network schedulers which enable the operating system to provide such support.

2. CKRM Framework

CKRM defines a class as a dynamic grouping of tasks (Linux equivalent of process threads) with the same resource allocation priority. The classification of tasks into classes is driven by a classification engine that is invoked at relevant task transition events (exec, setuid and the newly introduced settag). Once a task is classified, all subsequent resource requests generated by the task are identified with the class and provided differentiated service by the class-aware CPU, memory and I/O resource schedulers (also called controllers) included in CKRM. Incoming network connections for the class are differentiated by CKRM's socket queue network controller.

The main components of CKRM are shown in Figure 1:

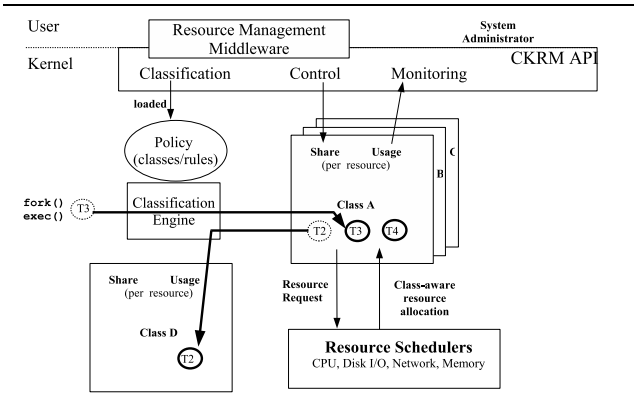

Figure 1. CKRM framework and life cycle

- **CKRM Core**: defines the data structures for class definitions and an API for a) registering various class aware schedulers independently; b) setting/getting the class shares and c) providing the interface and callback hooks into a loadable classification engine.

Proceedings of the International Conference on Autonomic Computing (ICAC'04)
0-7695-2114-2/04 $20.00 © 2004 IEEE

- **Classification Engine**: is an external policy driven kernel module. Its sole task is to classify tasks into classes.
- **Class-aware Resource Schedulers**: These are patches to the various resource schedulers (CPU, mem, disk, net) to make them class aware and enforce specific per class resource usage.
- **Resource Manager**: entity which determines the proportions in which resources should be allocated to classes. This could be either a human system administrator or a resource management application middleware as envisioned in the autonomic computing framework.

At system initialization time, the class aware schedulers and the classification engine register with the Core. CKRM provides a policy-driven Rule Based Classification Engine (RBCE). As inputs, RBCE accepts *policies*, consisting of a set of class definitions and a set of rules that associate task attribute values (e.g. executable, uid, gid, tag) with a class. Once a policy is loaded into RBCE and activated, it is used on important task events such as fork, exec, setuid and set_application_tag to classify tasks into the classes. Tags are a newly added attribute of the task, opaque to the kernel, and set by the application or a trusted user level daemon to assist in the task's classification based on the work being done by it. Policies can be dynamically changed or replaced and provides a clean separation of policy and enforcement. Each time a policy is changed, all existing tasks in the system are reclassified using the new classes defined. The CKRM infrastructure is flexible enough to accept classification engines other than RBCE.

As part of the initial policy or later, per-resource shares are assigned to each class in the system by the Resource Manager. Each class gets a separate share for CPU time, resident page frames, I/O bandwidth and incoming network (socket accept queue) connections. The resource schedulers then differentiate between requests from classes based on these shares. The schedulers and statistic gathering components of the kernel also maintain usage statistics for each controlled resource on a per-class basis. This allows Resource Managers to implement an adaptive feedback loop using class shares as activators and class usage statistics as sensors.

3. Class-aware Resource Schedulers

We give an overview of the design and implementation of the resource controllers. More details are available in [2, 1]. CKRM implements weighted fair share queuing for cpu, memory, block I/O and inbound network control. Outbound network control can be effectively handled through the existing netfilter mechanisms and do not require extensions in the Linux kernel. One of the overriding goals was to provide class based functionality with small extensions/modifications to the existing schedulers. This allows CKRM to take advantage of continuing improvements, by the open-source development community, in the base, class-unaware schedulers.

The current CPU scheduler (aka O(1)) provides a run queue for each CPU and does scheduling within each run queue. Occasionally or during idle times, the queues are load balanced to provide some degree of fairness. In CKRM, we provide a run queue per class/per cpu and deploy a simple two level scheduling and load balancing scheme. In the first level we perform a weighted fair share scheduling of classes and within the selected class we deploy the same scheduling behavior as the current scheduler. In load balancing we balance classes such that global class shares are achieved and within classes we balance again using the default mechanism to achieve intra class fairness.

The memory scheduler achieves its share controls by associating each page frame with the allocating task's class and with modifications to the page replacement algorithm. Instead of following the default LRU-like policy strictly, victim pages are preferentially chosen from classes over their allocation. This leads to memory allocations approaching the desired shares gradually and only when overall system utilization of memory is high.

CKRM's disk/block I/O scheduler creates explicit per-class queues. I/O requests submitted by tasks go into the queue of its current class. The scheduler then picks requests from the queues in proportion of the class shares and submits them to the low level device drivers.

Finally, the inbound network control associates network subclasses within each service class. Using netfilter packet marking techniques it tags inbound connect packets based on $< src, port, dst, port >$ classification with the network sub-class tag. The accept queues associated with individual sockets are then drained during `accept()` based on the share settings of said subclasses.

The performance data for the schedulers is available at [1]. Overall we have demonstrated that an operating system can achieve effective, efficient and scalable class-based resource control with minor/acceptable changes to its schedulers and thus can provide meaningful abstractions to the higher level workload management levels.

References

[1] Class-based kernel resource management. http://ckrm.sf.net/.

[2] 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.