CSC2227: Advanced Topics in Operating Systems Design

Implementing Resource Containers in K42
– Literature Survey –

David Tam, Adrian Tam, Reza Azimi

Feb. 28, 2003

1 Introduction

A problem in current operating systems is that the model of resource accounting does not necessarily reflect actual usage patterns. A portion of CPU usage charged to a user process may be consumed by preemptive system tasks such as interrupt handling. Moreover, a single user application may cross several resource domains (processes), which deceives the process-based resource accounting model. Without an accurate model of resource accounting, quality of service (QoS) guarantees cannot be provided. In addition, the system is vulnerable to denial-of-service (DoS) attacks.

In this report, we briefly review some of the more important efforts that have been made to address the problem identified above. However, we mainly focus on approaches that use resource containers as the basis for resource accounting and resource management. Then we highlight issues and provide ideas on how to incorporate the resource container model into the K42 operating system architecture.

2 Related Work

2.1 Resource Containers

Resource containers, as specified in [2], have addressed the problem of accurate resource accounting and management in uni-processor environments. A resource container is a logical entity that encapsulates a system resource and can be passed between different resource domains. All overheads associated with a service that manipulates several resources will be charged to the corresponding resource containers. Banga et al. [2] demonstrate that by abstracting system resources into containers, QoS can be guaranteed for high priority clients as well as mitigating various types of DoS attacks.

Lazy Receiver Processing [3] attempts to address the problem by identifying received packets with an associated service. This is accomplished by de-multiplexing the packet based on its socket, thus allowing the system to charge the overhead to the service that uses the socket. A limitation of this approach is that lazy receiver processing can only provide accurate accounting for network services. In fact the resource container model is a generalization of this idea that accounts for all system resources.

2.2 Cluster Reserves

Cluster reserves [1] extends the idea of resource containers to cluster-based network servers by considering resources within a cluster, known as cluster reserves. This paper provides an important lesson – good overall provisioning depends upon good resource usage on individual nodes. In the case of K42, we may substitute the word “nodes” with “processors”. This paper demonstrates that resource containers can be applied to cluster-based systems, resulting in the ability to provide QoS guarantees. Our project, however, is different from cluster reserves since cluster reserves are intended for cluster-based systems. Since communication costs are much higher in cluster-based systems than in multiprocessor systems, there may be a mismatch in the granularity of dispatching work. In addition, this work does not address the impact of resource containers on system scalability.

2.3 Scout

The Scout system [5] provides a different method of resource accounting. Scout identifies resource usage based on path and protection domains. A path is a logical entity that abstracts an I/O channel, such as from the Ethernet layer to the TCP/IP layer to the file system layer. Protection domains provide the abstraction of a layer of service to a logical entity. The Scout approach is different from the resource container approach in that Scout cannot encapsulate different I/O channels into one single logical entity. This ability is needed because multiple I/O channels may belong to a single application and it would be more logical to charge the resource usage to the application as a whole.

2.4 Multiprocessor Extensions

There have been attempts to address the issue of resource accounting in a shared-memory multiprocessing environment. Verghese et al. [7] propose abstracting system resources into a logical entity known as a system performance unit (SPU). SPUs associate a resource to a group of processes that provide a single service. For each service, the CPU usage time is measured by counting the number of time-slots the service requires in a time-sharing system. Memory usage is also measured and controlled during page-faults caused by the service. Since a service may not use all of its allocated resources, it is useful to have hard and soft limits in a shared-memory multiprocessors environment. Other services would then have the ability to borrow resources from an idle service. The borrowing may be revoked if the borrowed resource is needed by its rightful owner.

We believe that Verghese et al.'s approach is not sufficient. The SPU abstraction does not naturally extend to threads. As a result, resource accounting would be inaccurate when different threads within a process provide different logical services. In addition, an SPU does not address services provided by the kernel. Instead, in this approach, all kernel services are encapsulated into a single kernel SPU. Consequently, resources consumed by the kernel on behalf of users (e.g. services provided by preemptive system tasks) are not charged to the actual users.

2.5 The Virtual Machine Approach

An entirely different approach to the problem of resource containment is to make use of a virtual machine (VM). At one extreme, each application executes in an exclusive operating system on an exclusive VM. VM are multiplexed on top of a physical machine. It is then the responsibility of the virtual machine monitor (in IBM-speak, the hypervisor) to manage all physical resources. VMs can thus account for CPU usage as well as memory, network, and disk usage, to name a few. Such a solution is below the operating system level and may be quite coarse-grained. Requiring that each application or operating system be executed in a separate VM increases the overhead of communication and co-ordination, and may severely reduce desired sharing capabilities. Scenarios exist where is it desirable to run all applications within a single operating system while still providing a certain level of resource containment. Resource containers allow for more fine-grained control, whereas virtual machines offer coarser-grained control.

The Denali isolation kernel [8] is an example of recent work on VMs. It emphasizes the configuration of one application per VM, and goes further by reducing the role and capability of the operating system that is sandwiched in-between. Standard operating systems cannot be run on top of Denali, only simplified sub-operating systems may run. In addition, in their paper [8], they considered only independent applications that did not require sharing amongst each other. Disco and Cellular Disco [4] are virtual machine monitor architectures that address resource management on shared-memory multiprocessors, which matches the K42 target hardware. Disco claims to offer fine-grained resource sharing and addresses the granularity issues mentioned previously.

We realized that the VM approach is only a partial solution. VMs solve the problem of accurate accounting of more than just CPU usage. It can accurately account for memory and network bandwidth usage as well. However, the current VM technologies do not provide transparent transfer of the resource limits across VMs. Assuming there is separate VM for each application and that these applications wish to invoke each other's services, a resource container framework on top of the VM core is necessary to allow resource accounting to flow between VMs.

3 Resource Containers in K42

Our project will extend the idea of resource containers to an object-oriented shared-memory multi-processor operating system. This work will provide for a detailed understanding of how to provide accurate resource accounting in a scalable manner. The idea and necessity of resource containers in K42 has been implicitly suggested in [6]. Three approaches to CPU accounting are considered during an IPC call. The first approach is to not attribute execution time to the callee. The second approach is to attribute execution time to the caller, hinting at some of the ideas of resource containers. Unfortunately, this approach may lead to priority inversion. This approach was abandoned because the solution would be quite heavy-weight. The third approach is a hybrid of the first two approaches. CPU usage time is attributed to the caller until priority inversion points are reached. Subsequently, CPU usage time is attributed to the callee for the remainder of the IPC call. We are attempting to derive a better solution to the second approach that is based on the idea of delayed attribution. Perhaps we can simply keep track of the CPU time spent in the IPC call and later attribute it back to the caller once control has returned to the caller.

References

[1] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *ACM SIGMETRICS 2000*. ACM, 2000.

[2] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, Louisiana, February 1999. USENIX.

[3] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *2nd Symposium on Operating Systems Design and Implementation*, pages 261–275. USENIX, October 1996.

[4] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *17th ACM Symposium on Operating Systems Principles*, Kiawah Island, South Carolina, December 1999. ACM.

[5] O. Spatscheck and L. Peterson. Defending against denial of service attacks in Scout. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, Louisiana, February 1999. USENIX.

[6] K42 Team. Scheduling in K42. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 2001. `http://www.research.ibm.com/K42/white-papers/Scheduling.pdf`.

[7] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192, San Jose, California, October 1998.

[8] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, December 2002. USENIX.