Implementing Resource Containers in K42

Adrian Tam, David Tam, and Reza Azimi
March 18, 2003

1 Introduction

We have realized that integrating resource containers (RCs) into the K42 operating system framework is not an easy task. We have held a large number of long design sessions. We have realized that there are many complex scenarios; some should be accounted for in our design, while others should not be accounted for. We encountered a large amount of complexity and interactions of the various system components. There are trade-offs to be made between the aspects of performance, functionality, and flexibility. We wish to develop an elegant, simple solution that is fast (in terms of performance), flexible, and has adequate functionality to make it useful.

We spent a large amount of time on determining how to integrate CPU usage resource containers. We have realized that this is a very large task. Thus, we have only briefly attempted to determine how to integrate memory consumption and network bandwidth resource containers. We believe that the CPU usage resource container is the most important and complex of the three types to implement. Originally, we planned to implement resource containers for CPU usage, memory consumption, and I/O (e.g. disk and network) bandwidth. We may need to scale back our goal due to complexity and amount of work necessary for each type of resource container. However, we will try to address the design challenges of building resource containers fully in K42 in as much detail as we can.

Implementing resource containers touches on several themes of the course. This project exposes us to the internal design of an object-oriented research operating system that is fully functional with real applications, and therefore, it is extremely complex. Such a project would require deep and detailed understanding of the major system components including the general architecture of the system, CPU scheduling (especially in a multi-processor environment), memory management, and I/O structure. We are also exposed to system evaluation and benchmarking issues. This project involves topics such as concurrent programming, synchronization and deadlock detection/avoidance, and dealing with large, complex systems. We are designing, constructing, and evaluating the effectiveness/impact of the resource container mechanism under the context of a multiprocessor operating system designed for scalability. We have already encountered many problems, and just within the integration design. We hope to overcome these problems by the end of the project, thus learning many valuable lessons about dealing with complex systems.

2 K42 System Architecture Overview

In this section, we introduce important K42 components and their structure. These components play a key role in integrating resource containers. We will describe the K42 inter-process communication (IPC), scheduling, and memory management mechanisms.

2.1 Inter-Process Communication

K42 is based on a micro-kernel design, thus making heavy use of IPC mechanisms to communicate among various system servers. The K42 inter-process communication mechanism is called the protected procedure call (PPC). It is a light-weight facility that is heavily optimized due to the fact that K42 is a micro-kernel operating system. The model of inter-process communication is that it resembles a procedure call that
crosses protection domains (i.e. address spaces). A special and important feature of PPCs is that the calls all remain on the current physical processor to maintain locality and reduce cache-conflicts.

When a calling thread issues a PPC, it raises an exception and causes the K42 exception-level layer to be invoked. The caller thread is suspended at this point and a new thread is created in the target address space to handle the call and to execute the specified PPC function/method. K42 then exits the exception-level layer and enters the callee’s address to execute the newly created thread. Upon call completion, the callee thread is destroyed, the K42 exception-layer is invoked, and the caller thread is given return parameters/values and is resumed.

Parameters are passed using three possible methods (1) using 8 processor registers, (2) using one physical memory page that is mapped to the caller and callee’s address space, and (3) creating a shared-memory region for large quantities of data transfer. The most efficient and most commonly executed path is with the use of the 8 registers. To ensure that resource containers do not impose excessive overhead, we plan to reserve 1 of the 8 registers for passing a resource container ID.

2.2 Scheduling

K42 uses a two-level scheduling scheme. At the bottom level, there is a kernel-level scheduler that chooses a dispatcher, which is in fact a user-level scheduler, to run. At the top level, the dispatcher arbitrates CPU cycles it has been given among the user-level threads. The main advantage of this two-level scheduling design is that it reduces the frequency of interaction with the kernel. User-level threads can interact with the user-level scheduler as frequently as desired without necessarily dropping down to the kernel. The two-level scheduling design essentially provides a buffering layer that is beneficial. However, as will be described in the design section, this two-level scheme is a source of complexity for our design of CPU usage resource containers.

Also in K42, there is the notion of a resource domain which is orthogonal to address space and can span across several address spaces. Each resource domain might include several dispatchers that are equally scheduled with respect to the resource domain. Each resource domain belongs to one of four scheduling classes in the system\(^1\). Resource domains belonging to the same scheduling class are treated equally.

2.3 Memory Management

K42 uses a hierarchical design to manage memory. The layers and components that are important to resource containers are the page managers (PMs) and file cache managers (FCMs). A PM controls the allocation of physical pages to FCMs. In turn, an FCM controls allocation of its shared of physical pages to files. Loosely speaking, the concept of files is used refer to normal files on disk as well as the virtual address space of an application that is used for computation/execution storage. This virtual address space can be thought of as having portions temporarily stored in a file on disk when these portions are paged out (a.k.a. backing store).

Important memory management mechanisms that must be modified to integrate resource containers are the page fault handler and the page replacement mechanism. For example, consider an application that has reached its maximum allowed page allocation specified by its resource container. Upon the next page fault, one of the application’s existing pages must be recycled to fulfill the page fault rather than allocating a new physical page and exceeding page limits of the application. The FCM is the component that handles page faults for a corresponding file. The PM is the component that performs page replacement.

3 Design and Implementation

In this section we describe the major structural modifications that are required in order to implement resource container functionality into the K42 operating system. One solution would be to extend the existing implementation of the resource domains in K42 to provide the necessary functionality of K42. Although we have not yet ruled out this alternative, it seems resource domains are designed for different purposes, and therefore the amount of required changes in the system may be significant. Next, we describe the major system scenarios involving RCs.

\(^1\)The four scheduling classes are: system & hard real-time, gang-scheduled, soft real-time, general purpose, and background.
3.1 RC Creation and Registration

Each user must be entitled to one or more RCs to be able to run a task on the system. Each RC contains the resource share for each physical resource (e.g., processor, network interface, etc.) in the system. In particular each RC defines: (1) the CPU share percentage for each processor in the system; (2) the maximum physical memory size (in terms of MBytes); (3) disk I/O bandwidth share for each I/O interface (in terms of MBytes/second); and 4) network I/O bandwidth share for each network interface (in terms of MBytes/second). Therefore, under such a model, we can differentiate between the case where a task is given 50% of four processors and the case where a task is given 100% of two processors.

Resource shares, as mentioned in the RC, are guaranteed to be allocated to the application only if the application itself is able to use them. For instance, if an application suspends itself due to a dependence on resources with unbounded delay in availability (e.g., user input or I/O events), then part of its CPU share may be dynamically allocated to other tasks that are ready to execute.

RCs are created by the system superuser. There should be a certain admission control mechanism, either static or dynamic, to ensure the prevention of unregulated overbooking of system resources. This is in fact a hard problem to solve and has been the subject of research. For the moment, we assume that such a mechanism exists so that the total amount of physical resources allocated in the RCs do not exceed the actual system resources.

Each RC is registered in a server called the RC Server. An RC Identification Number (RCIN) will be issued upon registry. The RCIN is used by the applications to bind to a specific RC, and therefore, it must be designed to be difficult to forge by malicious users. Currently, we assume a simple 32-bit random value for RCINs in order to reduce the costs associated with RC lookup. We use a hash table in the RC server to lookup an RC given its RCID.

3.2 Binding Threads to RCs

The major goal of binding RCs to threads is to ensure that the set of threads that are running in the context of an RC are guaranteed to be given the resource share they are entitled to. We have decided to bind RCs to the dispatchers that are visible by the kernel and hence offer more flexibility in allocating resources to them. This means all of the user-level threads that share the same dispatcher would share the resource shares of the RC that is bound to the dispatcher.

There are three major scenarios of binding RCs to system activities: (i) launching a new application in the form of one or more processes, (ii) re-binding RC explicitly, and (iii) issuing a PPC. In this section we describe the first two, while the last scenario will be explored in the next section.

When a user with a given an RC launches a new process, the main dispatcher of the new process will be bound to the RC. The binding happens as a part of loading and spawning the new process. This dispatcher-to-RC binding remains valid until the program explicitly requests a re-binding to another RC. Figure 1 shows the major components of the system that are involved in the basic binding scenario. There are two types of usage of RCs in our system: (i) using RCs to do resource accounting accurately, and (ii) incorporating the RC mechanism in resource allocation and scheduling in order to actually guarantee some quality-of-service. Next, we describe how we implement each of these more in detail.

3.2.1 Resource Accounting

Once the binding between a dispatcher and an RC is established it is easy to account the resource consumption of an activity. In order to accounting the CPU usage of an activity accurately, every time slice that is given to a dispatcher would be charged the the RC that is currently bound to. Therefore, if a dispatcher re-binds itself from one RC to another (e.g., in the case of performing tasks of different users), the CPU that is allocated to each RC is accurately computed.

Given current hardware architectures, there is no easy and efficient way to measure the amount of memory that a dispatcher actually uses, since most of the dispatcher’s memory accesses are done transparently to the operating system. As an approximation, we take the number of page faults a dispatcher causes as the dispatcher’s memory usage figure. Therefore, at each page fault, we have to first identify the dispatcher who caused the page fault, and then charge the page-fault to the RC the dispatcher is currently bound to. The advantage of this approach is that it is easy to implement. However, it might not be precise enough in some situations. For instance, if a dispatcher only initializes memory that is subsequently used by the other
dispatchers bound to different RCs, all of the memory usage would be charged to the initializer dispatcher which is not a desired result. However, we would argue that such problematic situations can easily be eliminated by carefully designing the applications.

Figure 1: The general design of the system including the RC server. Thread 1 is bound to RC 1 throughout its execution. Also since thread 1 runs on the dispatcher of its address space, the dispatcher is bound to the RC 1. The kernel-level dispatcher queue is organized according to the CPU share values stored in the RC to which the dispatcher is bound.

### 3.2.2 Resource Allocation and Scheduling

In the current implementation of scheduling in K42, the dispatchers in the same scheduling class (resource domains) have equal priority to be scheduled. However, by binding RCs with different CPU share to different dispatchers in the same resource domain, we expect they are prioritized so that the get CPU proportional to their share designated in RC. We add a set of counters to the dispatcher descriptor to reflect the resource usage rate of the dispatcher. These counters are checked against the resource shares the bound RC is designated to use.

For CPU, the dispatchers are prioritized in the reverse order of their remaining CPU time slice. The counters inside the dispatchers would be reset at every fixed interval time \( T \). During the interval, giving a time slice to the dispatcher would result in increasing the counter. The remaining number of time slices would be calculated based on the value of this counter and the CPU share that is designated in the RC bound to the dispatcher. The more remaining number of time slice a dispatcher has, the higher its priority to be scheduled within a scheduling class.

For memory, the bottom line is that we have to stop the application from allocating new physical pages, once it hits the maximum number of pages that it can allocate according to the bound RC. In order to proceed, the application must be forced recycle one of the physical pages that is already allocated for the current FCM and, performing a page replacement. However, we have not explored the implementation details of such a scheme yet.

Another major issue is that an RC can be bound to more than one dispatcher. Therefore its resources
must be shared among such dispatchers. There are several ways to design such a sharing scheme. However, for the time being, we take the simplest and divide the resources of the RC equally among all of the dispatchers it is bound to.

### 3.3 Implementing PPC

In a PPC, the RC of the caller dispatcher would be transferred to the callee dispatcher. The transfer and re-binding of the RC is either explicit through an application visible normal parameter of PPC, or implicit and automatic through the PPC mechanism. Since PPCs are synchronous, in a PPC the caller thread is blocked until the call returns. However, other threads in the caller address space may be active, and therefore the RC would be bound to two dispatchers (or more) in two different address spaces. Upon return from the PPC, the RC has to be automatically revoked from the callee dispatcher by the PPC mechanism. Figure 2 shows the basic PPC implementation.

Since multiple PPCs can be active at the same time in an address space (e.g. in a server application) a single dispatcher may be simultaneous bound to more than one RC. The problem is how to distinguish the resource usage pattern of two user-level threads that are sharing the dispatcher but use different RCs. One solution is to explicitly re-bind RCs at every user-level context switch. The advantage of this approach is that it is simple to implement and would result in precise accounting and resource management. However, the major problem of this approach is that the cost of re-binding might become too high, if the user-level thread context switch occurs very frequently. Another approach would be to use distinct dispatchers in the callee address space for different PPCs. The problem is that the large number of concurrent PPCs to a server application might result in too many dispatchers which are expensive for the system to manage. The last solution is to assign the dispatcher the union of the resources designated in the two RCs, and rely on the user-level scheduler to allocate the resources to the threads according to their share. Therefore, the kernel must inform the user-level scheduler about the proportion of the RC a thread that is newly created for PPC, is entitled to use. This method would only work for system servers that are trusted. Otherwise, there is no guarantee that the user-level scheduler is not changed by the user application so that the incoming RCs could be exploited for the server program’s internal uses.

### 4 Division of Work

We plan to divide work among group members using a global work queue. Here is a list of tasks that can be executed in parallel. Each group member will execute a task to completion and then select another task from the global queue below.

1. Modifying the IPC mechanism.
2. Modifying user-level & kernel-level schedulers.
3. Developing a resource container server.
4. Modifying the FCM and PM components (memory management system).
5. Implementing resource containers as clustered-objects.

### 5 Logistical obstacles

No major logistical obstacles have been encountered. Since K42 is an open source operating system, we were able to obtain the source code and its related documentation easily. In addition, our group was able to obtain both the hardware and the software-based simulator for our project.

### 6 Evaluation Plan

There are a number of milestones in the project. Each of these milestones is evaluated by a series of benchmarks and micro-benchmarks. The benchmarks and micro-benchmarks will be tested on real hardware.
6.1 CPU Accounting

The first milestone in the resource container project is to accurately account for CPU usage in K42. The evaluation is divided into functionality testing and performance testing.

6.1.1 Functionality Test

The functionality test is divided into four micro-benchmarks. Each micro-benchmark addresses different usages with resource containers.

- This micro-benchmark measures whether CPU usage is accounted accurately in the absence of PPC calls at the process level. First, a single CPU intensive process, that does not make any PPC calls, is run and its runtime is measured. Examples of CPU intensive processes include finding prime numbers and finding the square root of large numbers. Next, the same CPU intensive process is run and is measured in the presence of another CPU intensive workload. This micro-benchmark is considered a success if it can be shown that the ratio of CPU consumption, as reported in the CPU resource container, is inversely proportional to the runtime of the process.

- The second micro-benchmark verifies whether CPU usage is measured accurately in the presence of PPC at the process level. First, a single CPU intensive process, which calls another CPU intensive process through synchronous PPC, is run and its runtime is measured. Next, the same process is run and measured in the presence of other CPU intensive workload. The micro-benchmark is considered a success if it can be shown that the ratio of CPU consumption reported in the CPU resource container is inversely proportional to the sum of the two processes’ runtime.

- The third and fourth micro-benchmark is similar to the first and second micro-benchmark, respectively. The only difference is that and I/P intensive process is used instead of CPU intensive process. These
6.1.2 Performance Test

The performance micro-benchmark’s setup is similar to the functionality micro-benchmark. There are four micro-benchmarks that measure the overhead of resource containers. The four micro-benchmark measure processes without PPC, processes with PPC, threads without PPC and threads with PPC. The overhead is measured by comparing the runtime increase in the presence of resource containers. Since the resource container’s overhead is expected to be small, the test is repeated 100 times to reduce measurement errors. Furthermore, to reduce the effect of cache miss and page faults, the test is pre-run 25 times to warm the cache and page table.

6.2 Scheduling with CPU Usage Resource Containers

The second milestone is integrating CPU scheduling with resource containers. This milestone is evaluated by functionality and performance test.

6.2.1 Functionality Test

There are four micro-benchmarks that measure the functionality of CPU scheduling using resource containers. The micro-benchmarks address both process and thread-level applications. In addition, it also addresses the complication associated with PPC calls.

- The first micro-benchmark measures whether quality of service is provided at the process level in the absence of PPC calls. First, a single CPU intensive user process is assigned a resource container that reserves CPU usage (ie: 50%). This process is run and runtime is measured. Next, the same CPU intensive user process is assigned a resource container that reserves less CPU (ie: 25%). The benchmark is considered a success if the user process that has a higher CPU usage reservation runs twice as fast as the case with less CPU reservation.

- The second micro-benchmark measures whether quality of service is provided at the process level in the presence of PPC calls. First, a single CPU intensive user process, which calls another CPU intensive user process through synchronous PPC, is assigned a resource container that limits CPU usage. This process is run and runtime will be measured. Next, the same CPU intensive user process is assigned a resource container that reserves half the CPU. The benchmark is considered a success if the user process that has a higher CPU usage reservation runs twice as fast as the case with less CPU reservation.

- The third and fourth micro-benchmark are similar to the first and second CPU scheduling functionality micro-benchmark. The only difference is that threads, instead of processes, are assigned CPU usage. Consequently, thread level scheduling with CPU usage resource container is measured.

6.3 Scheduling with CPU Usage Resource Containers

6.3.1 Performance Test

The performance test measures the overhead of using CPU resource containers in scheduling processes and threads. First, the runtime for a CPU intensive process, which is not assigned to any resource container, is measured. Next, the same CPU intensive process is assigned a resource container that limits CPU usage by 50%. The overhead is the difference between twice the runtime that does not have use any resource container and the runtime that limits CPU usage by 50%. This micro-benchmark is repeated for processes with PPC call and thread-level applications. The test is ran 100 times and pre-runs 25 times to maintain measurements accuracy.
6.4 Multiprocessor Issues with CPU Resource Containers

Since the CPU resource usage is accounted per physical processor, we do not expect any functionality or performance problems. In order to verify this hypothesis, the functionality and performance micro-benchmarks will be measured in a multiprocessor environment. The difference is that one test process/thread will be run per physical processor.

6.5 Memory Usage Accounting

Memory usage resource containers will be evaluated in two steps. First, the micro-benchmarks will verify memory usage is accurately measured. Next, the micro-benchmarks will verify the scheduler uses memory resource containers correctly.

6.5.1 Functionality Test

The functionality test examines whether page usage is accurately reflected in the resource container. First, a memory intensive user process, which does not call other process through PPC, is run and its memory usage (reported as pages) is accounted through a resource container. A process that allocates and traverses a large array is an example of memory intensive user process. Next, the reported memory usage is compared to the result reported by K42’s region. K42 already provides memory accounting within a single process through the region object’s method. The benchmark is gaged to be successful when the two results are identical. Memory usage, in the presence of PPC, can be measured at the process level. A memory intensive user process, that calls another memory intensive user process through PPC, is run and its memory usage is measured. The benchmark is considered a success if the resource-container reported memory usage is identical to the sum of the two processes’ memory usage (as reported by the Region object). The thread-level results, with and without PPC, can be measured similarly.

6.5.2 Performance Test

The performance test measures the overhead associated with the memory usage. First, caches are flushed so that an accurate count of page faults can be maintained. Next, an user process, which does not call other processes through PPC, allocates and traverses large array. The runtime for this operation is measured for both with and without the presence of resource container. The case for process-level with PPC calls and thread-level memory resource containers can be measured similarly.

6.6 Scheduling with Memory Usage Resource Containers

Memory usage resource containers can be evaluated with respect to both functionality and performance.

6.6.1 Functionality Test

The functionality test verifies that the number of pages owned by a resource container accurately conforms to the user specified value. First, a memory intensive process is assigned a memory resource container that limits the number of pages (ie: 50 pages). Next, the number of pages owned by the memory intensive process is examined by invoking the Region object’s method. The micro-benchmark is gaged to be successful when the number of pages allocated to the process equals to the specified values in the memory resource container. Memory usage, in the presence of PPC can be measured similarly. The difference is that the memory intensive process calls another memory intensive process through PPC. The benchmark is gaged to be successful when the total number of pages allocated to the two processes equals to the value specified by the memory resource container. The case for thread-level resource containers can be measured similarly.

6.6.2 Performance Test

The setup of the performance test is similar to the functionality test scheduling with memory usage resource container. The difference is that the cache is flushed in the beginning, and the runtime of the processes and threads are measured. The performance test measures the overhead of scheduling using memory usage resource container.