Predictable Management of System Resources for Linux

Mansoor Alicherry* K Gopinath†

Department of Computer Science & Automation

Indian Institute of Science, Bangalore

Abstract

In current operating systems, a process acts both as a protection domain and as a resource principal. This may not be the right model as a user may like to see a set of processes or a sub activity in a process as a resource principal. Another problem is that much of the processing may happen in the interrupt context, and they will not be accounted for properly. Resource Containers[1] have been introduced to solve such problems in the large-scale server systems context by separating out the protection domain from the resource principal by associating and charging all the processing to the correct container. This paper tries to investigate how this model fits into a Linux framework, especially, in the *soft real time* context. We show that this model allows us to allocate resources in a predictable manner and hence can be used for scheduling soft real-time tasks like multimedia. We also provide a framework in Linux which allows privileged users to have their own schedulers for scheduling a group of activities so that they can make use of the domain knowledge about the applications. We also extend this model to allow multiple scheduling classes.

1 Introduction

A general purpose operating system has to manage the underlying hardware resources (CPU, memory, disk etc) to provide a satisfactory performance for a mix of interactive, batch and, possibly, real time jobs. The system may be acting as a server providing specialized services to other computers. For many users perceived speed of computing is governed by server performance.

Modern high performance servers (eg. Squid) use a single process to perform different independent activities. However an activity may constitute more than one process. This may be for fault isolation and modularity. An example of this is CGI processing in http servers. In these cases, the user may wish to control the scheduling and resource allocation for a group of processes together.

In current general purpose operating systems, scheduling and resource management do not extend to significant parts of the kernel[1]. An application has no control over the consumption of many resources that the kernel consumes on the behalf of the application. Whatever control it has is tied to assumption that each process is an independent activity.

In summary, in current general purpose operating systems, processes have the dual role of acting as a protection domain and as a resource principal. Hence, the notion of resource principal has to be separated from protection domain for better resource management. Resource containers[1], proposed by Banga, Druschel and Mogul, allow for fine-grained resource management by removing the role of resource principal from processes. Their work primarily concentrates on accounting correctly the overhead of network processing inside the kernel to the right resource container.

A resource container encompasses all system resources that a process or group of processes uses to perform an independent activity. All user and kernel processing for that activity is charged to the resource container and scheduled at the priority of the container.

In this paper, we study the management of system resources for Linux using resource containers. Though our design has many aspects similar to [1], our emphasis has been on the use of resource containers for scheduling soft real-time tasks[1] like multimedia. We extend the model of resource containers to support multiple scheduling classes. We also provide a framework in Linux for allowing different scheduling functions for different sets of applications. We also extend the APIs provided in the original model on resource containers. We also provide a */proc* interface to resource containers so that its parameters are easily accessible.

We have implemented *RCLinux*, a resource container

*The author is currently with Bell labs, Murray Hill, NJ. The author can be reached at mansoor@research.bell-labs.com

†The author can be reached at gopi@csa.iisc.ernet.in

[1]Soft real-time tasks do not have "strict" deadline requirements, though some timeliness requirements still exist.

implementation for Linux version 2.2.5-15 (Red Hat 6.0). The patch and detailed documentation is available at **http://casl.csa.iisc.ernet.in/~mansoor/proj**.

1.1 Scheduling Anomalies in Current Operating Systems

Traditional schedulers have evolved along a path that has emphasized throughput and fairness. Their goal has been to effectively time-multiplex resources among conventional interactive and batch applications. But today, there is a growth of applications like multimedia audio and video, virtual reality, transaction processing etc. whose resource requirements are real-time in nature. Unlike conventional requests, real-time resource request need to be completed within application specific delay bounds, called deadlines, in order to be of maximum value to the application. For conventional and real-time tasks to co-exist, a scheduler has to allocate resources in such a way that real-time processes should be able to meet their deadlines, interactive jobs get good responsiveness and batch jobs should be able to make some progress. This is a very challenging problem.

Unix provides the "nice" system call to reduce/increase the base priority of processes so that there is forward progress in an application mix. But the values for these nice calls are often non-intuitive and many experiments have to be done to come up with the correct values. And these values are highly dependent on the application mix and have to changed every time the application mix changes.

Another problem in the current general purpose operating system is that scheduling is done on a per process basis or on a per thread basis. A user having more number of processes/threads gets more CPU time than another user running a lesser number of threads. A malicious user can hog the CPU by creating lot of processes and hence preventing the progress of other users' processes. We need a mechanism for preventing these types of "denials of service" by guaranteeing a fixed percentage of CPU for the users if required.

Yet another problem in the current general purpose operating systems is that network-intensive applications do most of the processing in interrupt context. Processing in interrupt context is either charged to the process that was running when the interrupt occurred or it is not charged at all. This can lead to inaccurate accounting and hence inaccurate scheduling.

Network servers have become one of the most important applications of large computer systems. Network processing in UNIX is mostly interrupt driven. Interrupt processing has strictly higher priority than user level code. This leads to interrupt live-lock or starvation when there is a high network activity.

Another problem in current general purpose operating system is in the lack of support for real time processes. Unix SVR4[2] supports static priority real-time classes, but it has been shown to be of not much use[3]. Real time classes are supported by having a global priority range for each class, and real the time class has higher priority value than the system class, which in turn have higher priority than the time shared class. Scheduling is done based on this priority. So when there is a process in the real time class, none of the system and time shared class processes are allowed to run. This model is fine for hard real time tasks, where the cost of missing a deadline is really high and efficiency is not of much concern. But this is not the right model for soft real time applications like multimedia, where the results are not catastrophic when the deadlines are not met. Also real time processes may depend on the system processes for some system services, but they are not able to get those services because of their own presence.

Nieh et al[3] studied the scheduling in SVR4 in the context of an application mix of typing, video and compute jobs. They found that no combination of assignment of different priority and classes to these applications gives a satisfactory performance to all the applications. They also found that the existence of a real time static priority process scheduler in no way allows a user to deal with problems of scheduling this application mix. They found that when using the real time class, not only do application latencies become much worse than desired, but pathologies can occur due to the scheduler such that the system no longer accepts user input.

For example, when the time shared class was used for all of the three type of application, compute bound (batch job) tasks performed well. This was due to the fact that the batch application forks many small programs to perform work, and then waits for them to finish. Since this job sleeps, the TS scheduler assumes it as an I/O intensive "interactive job" and provides it repeated priority boosts for sleeping.

1.2 What Useful Scheduling Models Exist?

Three useful concepts have been identified independently for their effectiveness in scheduling for applications in restricted domains[4]: best-effort real-time decision making, exploiting the ability of batch applications to tolerate latency and proportional sharing.

Best-effort decision making combines earliest-deadline scheduling with a unique priority for each request to provide optimal performance in under-load and graceful degradation in overload. Multilevel feed-back schedulers, as typified by UNIX time-sharing, take advantage of the ability of long running batch applications to tolerate longer and more varied service delays to deliver better response time to short interactive requests while attempting to ensure that batch applications make reasonable progress. Proportional sharing, also known as weighted fairness, has long been advocated as an effective basis for allocating resources among competing applications. Here each application is allocated resources proportional to its relative weighting.

1.3 Brief Overview of the Linux Scheduler

Linux supports 3 scheduling policies: SCHED_FIFO, SCHED_RR, and SCHED_OTHER. SCHED_OTHER is the default universal time-sharing scheduler policy used by most processes; SCHED_FIFO and SCHED_RR are intended for special time-critical applications that need precise control over the way in which runnable processes are selected for execution.

A static priority value is assigned to each process and scheduling depends on this static priority. Processes scheduled with SCHED_OTHER have static priority 0; processes scheduled under SCHED_FIFO or SCHED_RR can have a static priority in the range 1 to 99.

All scheduling is preemptive: If a process with a higher static priority gets ready to run, the current process will be preempted and returned to its wait list. The scheduling policy only determines the ordering within the list of runnable processes with equal static priority.

There is a single run-queue. The scheduler goes through each process in the queue and selects the task with the highest static priority. In case of SCHED_OTHER, each task may be assigned a priority or "niceness" which will determine how long a time-slice it gets. The "counter" attribute of each task determines how much time it has left in its time-slice. The scheduler selects the task with highest counter value as the next task to run. After every task on the run-queue has used up its time-slice (counter = 0), the counter for each task is set to the original priority + half the counter. In this way "interactive tasks" (tasks which were not in run-queue but whose counter was not zero) get a priority boost.

1.4 Related Work

Lazy Receive Processing (LRP)[5], proposed by Druschel and Banga, solves the problem of inaccurate accounting and interrupt live-lock due to network processing in interrupt context by identifying the process that caused the traffic and doing the network processing in the context of that process. In LRP, network processing is integrated into the system's global resource management. Resources spent in processing the network traffic are associated with and charged to the process that caused the traffic. Incoming network traffic is processed at the scheduling priority of the process that receive the traffic and excess traffic is discarded early. Later on, the concept of resource containers[6], proposed by Banga, Druschel and Mogul, was introduced as a means of resource management in large-scale server systems. A prototype implementation was reported for Digital UNIX. A FreeBSD implementation of it is available in [7].

Goyal, Guo and Vin [8] propose an operating system framework that can be used to support a variety of hard and soft real-time applications in the system. This framework allows hierarchical partitioning of CPU bandwidth: the OS partitions the CPU bandwidth among various application classes, and each application class, in turn, partitions its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. They used Start-time Fair Queuing (SFQ) algorithm for such a partitioning.

SMART[4], proposed by Nieh and Lam, a Scheduler for Multimedia And Real-Time applications, supports both real time and conventional computations and provides flexible and accurate control over sharing of processor time. SMART is able to satisfy real-time constraints in an efficient manner and provides proportional sharing across all real-time and conventional tasks. When not all real-time constraints are met, SMART satisfies each real time task's proportional share, and adjusts its execution rate dynamically. SMART achieves this by combining the concepts of proportional sharing, latency tolerance, and best-effort real-time decision making. SMART uses an algorithm based on weighted fair queueing (WFQ) to implement weighted fairness. The concept of virtual time, biased with notion of latency tolerance, is used to measure the resource usage of the applications. The biased virtual time is then used as priority in the best-effort scheduling algorithm so as to satisfy as many real-time requirements as possible.

Resource Kernel[9], proposed by Rajkumar, Juvva, Molano and Oikawa, is another approach to real-time scheduling. A resource kernel is one which provides timely, guaranteed and enforced access to physical resources for applications. With resource kernel, an application can request the reservation of a certain amount of a resource, and the kernel can guarantee that the requested amount is available to the application. A reservation can be time-multiplexed or dedicated. Time-multiplexed reservation is represented by parameters C, D & T, where T represents recurrence period, C represents processing time required within T, and D is the deadline within which the C units of processing time must be available within T. Oikawa et al[10] have an implementation of resource kernel for Linux.

RTLinux (RealTime Linux)[11] is an extension to Linux that handles time-critical tasks. In RTLinux, a small hard-realtime kernel and standard Linux share one or more processors, so that the system can be used for applications like data acquisition, control, and robotics while still serving as a standard Linux workstation.

KURT (KU Real-Time Linux)[12] is another approach to support real-time in Linux. KURT Linux allows for explicit scheduling of any real-time events rather than just processes. It has two modes of operation, the normal mode and the real-time mode. In normal mode, the system acts as a generic Linux system. When the kernel is running in real-time mode, it only executes real-time processes.

2 The RCLinux Resource Container Model

Resource containers[1] encapsulate all the resources consumed by an activity or a group of activities. Resources include CPU time, memory, network bandwidth, disk bandwidth etc. Our focus in this paper has been on CPU time. [2]

Scheduling and resource allocation is done via resource containers[3]. Processes are bound to resource containers to obtain resources. This *resource binding* between a process and a resource container is dynamic. All the resource consumption of a process is charged to the associated resource container. Multiple processes may simultaneously have their resource bindings set to a given container.

A task starts with a default resource container binding (inherited from its creator). The application can rebind the task to another container as the need arises. For example, a task time-multiplexed between several network connections can change its resource binding as it switches from handling one connection to another, to ensure correct accounting of the resource consumption.

Tasks identify resource containers through file descriptors. The semantics of these descriptors are the same as that of file descriptors: the child inherits the descriptors from the parent, and it can be passed between unrelated processes through the Unix-domain socket file descriptor passing mechanisms. APIs are provided for operations on resource containers. Security is enforced as an application can access only those resource containers that it can reference through its file descriptors.

We have added a new name space corresponding to the resource container: the *resource container id*, similar to the *pid* for processes, as it was not possible to access some of the resource containers through the existing APIs (*e.g.* a container that has no processes directly associated with it). This name space is available to users having the right privilege through the */proc* interface.

Resource containers form an hierarchy (figure 1). A resource container can have tasks or other resource containers (called *child containers*) in its *scheduler bindings* (i.e. the set of schedulable entities). The resource usage of a child container is constrained by the scheduling parameters of its parent container. On the top of the hierarchy is the root container. This encapsulates all the resources available in the system.

Hierarchical resource containers make it possible to control the resource consumption of an entire subsystem without constraining how the subsystem allocates and schedules resources among it various independent activities. This allows a rich set of scheduling policies to be implemented. Our Linux implementation allows the scheduling policies to be changed dynamically through APIs. It also allows these policies to be in dynamically loadable modules. This allows privileged users to try out various scheduling algorithms and to use better ones with a specific application mix.

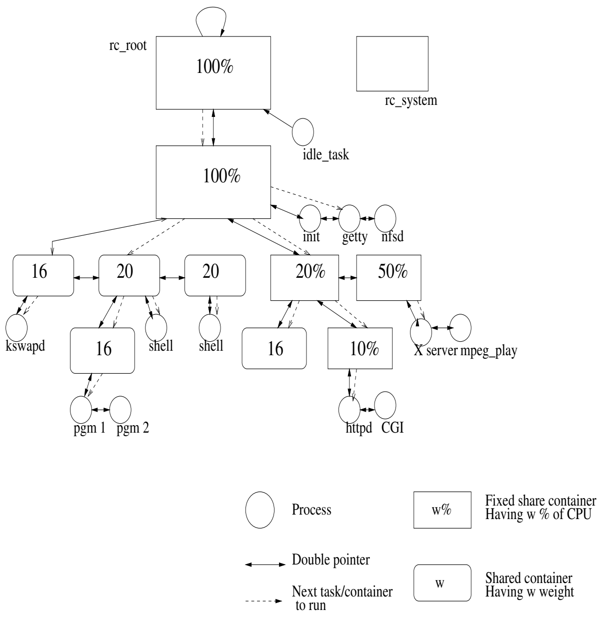

Figure 1: Resource container hierarchy

The CPU resources allocated to a container may be either a fixed share of the resource its parent container is entitled to (called *fixed share child container*), or it may be shared with other children of the parent (called *shared child container*). In the case of shared children, the amount of CPU time it is entitled to is proportional to the weight of the container relative to other shared child containers of parent.

Along with fixed and shared child containers, our implementation support multiple scheduling classes. Scheduling classes have strict priorities, i.e. a container in the lower priority class will not be scheduled in the presence of container with a higher priority class.

For scheduling soft real-time processes, we need to attach them to fixed share containers. The CPU reservation of these containers have to be at least the amount of processing required for those processes. Unlike hard real-time scheduling, this will allow other time shared processes to have reasonable progress if the CPU reservation of soft-real time process is not 100%. Higher scheduling classes should be used only when absolutely necessary (eg. hard real-time) as this can possibly make all the processes in the lower class starve.

Our design does not take care of any interrupt live-locks that can occur due to high network activity. It has been shown that LRP[5] gives stable throughput under high load, hence we plan to incorporate it at a later time.

[2] We collect the resource usage of resource containers for CPU time, network bandwidth and disk usage. This can be used for computing the priority of the resource container if appropriate weights are assigned to each of the resource usage. Also the network bandwidth usage will reflect on the CPU usage since the protocol processing for the packets are significant and we account for the network processing properly.

[3] The descriptions in this section closely follow those in [1] & [6].

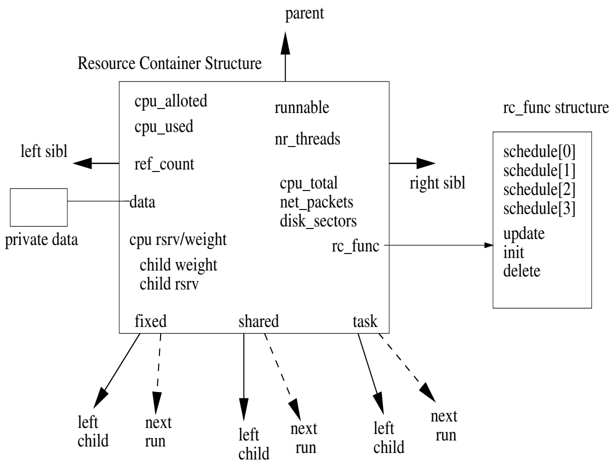

Figure 2: Resource container structure

2.1 Resource Container Data Structures

Resource containers are first-class kernel objects. They have various attributes that are required for scheduling and resource accounting along with functions for various operations on containers (figure 2). Some of the important attributes are:

1. *Mechanism Data*: This contains the necessary data of the container mechanism. This includes reference count, the list of runnable threads and next thread to run in the scheduler binding of the container, the list of child containers and next container to run in the fixed share children list and shared children list, and the parent container. The resource container hierarchy is implemented using a left-child, right/left sibling data structure, with each container having a pointer to its parent. The reference count keeps track of the number of references to the resource container. A reference to a resource container can be from a child container, from a task as its resource binding or from the global file table.

2. *Scheduler Data*: This contains data necessary for the scheduler. This includes the weight or CPU reservation that this container is entitled to from parent, sum of the CPU reservations and weights of the child containers, CPU time allocated to this container and CPU time used by this container if this container is in the *current path* (*i.e.* the path from root to the current container in the container hierarchy), number of runnable threads in the scheduler binding of the container, and a *runnable* bit mask. Each bit in the runnable bit mask indicates whether the container is *runnable* for the scheduling class corresponding to the bit. A container is said to be *runnable* for a class if there is a task in its scheduler bindings which belongs to that class or if it has a child container which is runnable for that class.

3. *Container specific functions*: Each resource container can have its own scheduler function for each of the scheduling classes and its own update, init and delete functions. The update function is used to update the scheduler data. This is called by the scheduling routine of the child container before it gives the control of the CPU back to its parent (subsection 2.3). The init routine is called when a resource container is created or when the container specific function is changed by calling *set_rc_functions()* (section 2.5). This function may be used to initialise the resource container variables or to allocate space for container's private data. The delete function is called when the resource container is destroyed. This may be used to free the space allocated for private data. Each resource container has a pointer to a structure containing these function pointers.

4. *Statistics*: Statistics includes the total CPU usage of the container, the number of disk reads and writes, the number of network packets sent etc. Statistics will be propagated to the parent container, when the container is destroyed, if a flag RCPROPSTAT (i.e. propagate statistics flag) is set in the container.

5. *Private Data*: This can be used by the container specific scheduler and update routines.

2.2 Global Data Structures

To fully characterise a system-level resource container abstraction, we need additional global variables such as:

1. rc_root: This is the root container. All the resources in the system are allocated to the root container and hence to the hierarchy rooted here. Any container or task that has to get a resource has to be in the tree rooted here.

2. rc_system: This container is used to charge all the processing that could not be directly associated with any container (eg. network processing for a packet that is not destined to any of the processes). This container is not part of the hierarchy rooted at rc_root. The statistics information in this container can be used by the system administrators to monitor the system.

3. rc_pool: Pool of resource containers from where a resource container structure is allocated. When a resource container is freed, the resource container structure is returned to the pool.

4. class_lastrun: When a task of higher class becomes runnable, it preempts the currently running task. When the task of higher class is no longer runnable, then we have to restart the scheduling with the container where we had left off. This value is stored in class_lastrun array. If the element of class_lastrun that corresponds to the highest runnable class is NULL, the scheduling starts from root.

5. rc_current: This points to the currently chargeable container. This cannot be taken as the container to which the currently running task is bound, since we are providing a facility for the task to change its binding on the fly through a system call and during the call there is a change.

2.3 Resource Container Scheduler Framework

Each resource container has its own scheduling function (*rc_scheduler*) for each scheduling class. This can be in a loadable module and can be set on the fly using the *set_rc_functions()* system call (see section 2.5). The *rc_scheduler* function is passed the resource container and the class as the arguments. It returns the task that is to be scheduled next. The class is passed as an argument since it can use the same scheduling function for each of the classes. Similar is the reason for passing the resource container.

When *rc_scheduler* is called, the resource container structure will have two of its variables *cpu_allotted* and *cpu_used* set. This invocation of the scheduler is allowed to allocate *(cpu_allotted - cpu_used)* amount of CPU time to its child containers which are marked runnable for that particular class or to tasks in its scheduler binding belonging to that class. When it is allocating a CPU for a child container, it sets the *cpu_allotted* value of that container to the required amount and returns by calling the class specific scheduler routine of child container. If it is allocating CPU for a task, then it sets *cpu_allotted* in the task structure of that process and returns the process. When *cpu_used* is greater than *cpu_allotted* it first calls the update function of parent, with this container as the argument, and then returns by calling the class specific scheduler function of the parent. The update routine updates the *cpu_used* of the parent by adding to it *cpu_used* of the child and sets *cpu_allotted* and *cpu_used* of the child to 0. Any priority re-computation that has to be done is also done in the update routine.

2.4 *rc_schedule()* routine

This is the default scheduler function for all classes for all resource containers. This is written conforming to the scheduler framework given above. The scheduling starts with root.

The resource container allocates CPU for the containers having fixed CPU share, by an amount proportional to the CPU reservation of the children. Any remaining CPU time is allocated to the tasks in the scheduler bindings of the container. After that it will allocate the remaining CPU time for the time shared child containers, by an amount proportional to their weights. At anytime, if the CPU usage of the container exceeds the allocated time, or all the runnable containers and tasks are given the CPU then the control is passed to parent container by calling the class specific scheduling function of the parent.

The root has a different scheduler function; it returns NULL. This is because once all the child containers and tasks are scheduled by the resource container, it will call the schedule function of the parent to give back the control of the CPU. But for root, this cannot be the case as we have to restart the scheduling of the whole tree. One way to achieve this is to check whether the container is root every time before calling the scheduler function of the parent container and restart the scheduling of the tree if it is root.

Another solution to this problem, which is cleaner and what we follow, is to have a different scheduler function for root. When the scheduler function for the container returns NULL, the *schedule()* function calls another routine *rc_root_schedule()* which restarts the scheduling of the tree, and returns the next thread to run.

rc_root_schedule() checks for the runnable flag of the root. If it is zero, then there are no processes in the system and it returns the *idle_task*. Otherwise it calls *rc_schedule()* with root and the highest runnable flag as the argument. If this function returns a task, that task is returned to *schedule()*. Otherwise it calls *rc_schedule()* again. This is because the root may be looking at some part of the tree (towards right) where there may not be any task in the highest class, so the first call to *rc_schedule()* will return NULL. The second call to *rc_schedule()* will restart the scheduling from the left most child and it will eventually find a task to run.

Similarly the update function for root is slightly different from update function of other resource containers, since the parent of root is root itself.

One important implementation issue that has a bearing on the *design* is that even if we are using regular tree traversal for scheduling, we cannot use any information on the stack between two context switches since the kernel stack changes for each context switch. So any information that is needed across context switches has to be kept in the resource container data structures itself. That is why *schedule()* calls *rc_schedule()* to find the next thread, rather than *rc_schedule()* itself acting as the sole scheduler function.

2.5 RCLinux Resource Container APIs

Resource containers can be accessed either through system calls or through the */proc* interface.

2.5.1 System Calls

System calls are provided for the users to make use of resource containers. Most of these system calls take the resource container descriptor as one of the arguments.

All the APIs except *set_rc_functions* were defined in [6]. But we have added the *pid* argument to *get_rc* and *set_res_binding* to make them more useful. *pid* was required as the argument to control the resources allocated to the processes which do not use the APIs (eg. an already existing application). This also gives the system administrator more control.

1. int rc_create(void)

   This creates a new resource container and associates a file descriptor for the process with it. It returns the descriptor.

2. int get_rc(int pid)

   This returns a descriptor for the resource container to which the process *pid* is bound. If *pid* is zero, the descriptor of the container of the calling process is returned. The permissions for obtaining the resource container descriptor of a process is the same as that of sending a signal to the process (ie. the calling process must either have root privileges, or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process). This will return a new descriptor every time this function is called.

3. int set_rc_parent(int rcd1, int rcd2)

   Sets the parent of the resource container corresponding to *rcd1* to that of *rcd2*. It also checks whether the resources can be allocated to the child container from the new parent before changing the parent. It changes the scheduling parameters of the old parent, the new parent and the child container to reflect the change. The child container is removed from the binding list of the old parent and added to that of the new parent. This is used to add a newly created container to the resource container hierarchy, or to change the structure of the hierarchy.

4. int set_res_binding(int rcd, int pid)

   Sets the resource binding of the process *pid* to the resource container corresponding to *rcd*. If *pid* is zero, the scheduler binding of the calling process is changed. Any future processing done by the process task will be charged to the new container. The task is removed from the scheduler bindings of the old container and inserted into that of the new container. Scheduling parameters of the containers are changed. The *need_resched* attribute of the task structure is set so that the task is preempted at the earliest safe point. The permissions to change the scheduler binding is the same as that of sending a signal.

5. int set_fd_rc(int fd, int rcd)

   This binds the open file or socket corresponding to *fd* to the resource container corresponding to *rcd*. All the future processing for the file will be charged to the new container.

6. int set_rc_opt(int rcd, struct rcopt *rco), int get_rc_opt(int rcd, struct rcopt *rco)

   Sets/gets the options of a resource container. Options includes scheduling parameters, statistics flags etc. *rcopt* is a structure that contains attribute type and attribute value as the fields.

7. int set_rc_functions(int rcd, struct rcfunc *rcfp)

   Sets the class specific scheduler functions and update function of a resource container. *rcfunc* is a structure containing the function names. This allows a privileged user to load a module containing the functions and call this system call to use these functions.

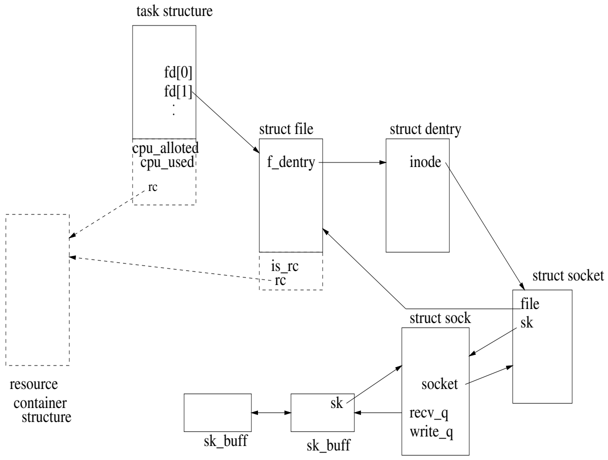

Figure 3: Kernel data structures (those dotted are new in RCLinux)

3 RCLinux Kernel Modifications and Redesign

The RCLinux task structure *task_struct* contains a pointer to the resource container it is bound to (figure 3). Whatever resources are allocated to the resource container, it allocates for the processes in its scheduler binding and to its child containers.

Processes can access the resource containers through file descriptors (resource container descriptors). A pointer to the resource container and a flag was added to the *file* structure. If the flag is 1, then the file structure corresponds to a resource container. Otherwise, it has the normal Linux semantics (i.e. file, socket etc).

3.1 Changes to the File Management Subsystem

When a file is open, the resource container pointer of the file structure is set to the container to which current task is bound. The reference count of the resource container is incremented for each reference of the file structure (this is different from the reference count of the file descriptor since two file descriptors may point to the same file structure). Similar processing happens for the *socket()* call also.

In the close routine, the reference count of the associated resource container is decremented if there are no more references to the file structure.

3.2 Changes to Network Subsystem

Most of the protocol processing for incoming packets and for the outgoing packets during retransmission happens in the bottom-half handler *net_bh()*. In the current Linux kernel, proper accounting of this processing is not done. The network subsystem has been modified to do proper accounting. The Linux kernel already allows us to work out which socket file descriptor a network buffer (*skbuff*) structure belongs to, so we can always find the resource container for a packet by looking up the correct socket file. The accounting has been done by recording the timestamp counter (subsection 3.7) at the beginning and end of the bottom-half handler.

If the resource container corresponding to a network packet cannot be identified (eg. a packet to a port which no process is bound to), then it is charged to a separate container *rc_system* which is not part of the hierarchy rooted at *rc_root* (subsection 2.2). The number of bytes sent and received is also kept in the resource container as part of statistics.

3.3 Changes to *schedule()*

Whenever a process has to be preempted, Linux sets a flag in the task structure of the currently running process. When control reaches one of the safe points (eg. return from a system call), the *schedule()* function is called to determine the next process to be scheduled and to context switch to that process. The *schedule()* function has been changed to support the resource container framework.

The *schedule()* routine updates the CPU usage of the current resource container by adding to it the CPU usage of the currently running process. Then it checks whether any process has become runnable in any higher class than that of the current container. If there is any, the highest class runnable resource container is the next resource container to be scheduled. Otherwise the current container is scheduled.

Once the next resource container is found, the class specific scheduler function of that container is called to decide on the next task to be run. This function need not necessarily return a task in its own scheduler binding since it could allocate the CPU to a child container and call its scheduler function, or its CPU time allotted could have expired by this time, in which case it calls the scheduler function of the parent to give back the control of the CPU (subsection 2.3).

If the class specific scheduler function of the next resource container to run returns NULL, then scheduling restarts from the root container.

After finding out which task to schedule next, the container to which this task is bound is saved. If the next task to be scheduled is not the currently running one, a context switch is effected.

3.4 Changes to *fork()* and *exit()*

Linux supports a *clone* system call that an user can use to create a clone of a process. A flag is passed as an argument to the *clone* call that specifies the type of the objects (eg virtual memory, signal mask etc) that are to be shared. Both fork and clone system call make use of the same function but use different flags. A new flag CLONE_RC has been added to the clone flags. If CLONE_RC is set then this function allocates a new resource container for the child process and makes it the child of the resource container of the parent process. Otherwise the resource container of child process is set to that of the parent process. When the child process is made runnable, it is added to the scheduler bindings of the resource container to which it is bound to.

When a process exits, Linux frees most of the resources (eg. virtual memory, file tables etc). This cannot free the complete task structure since some of the information is required later (eg. exit code for the parent, fields in task structure used by the scheduler, and the kernel stack since the process is still the "currently running" process). These fields are freed when the parent calls wait. A resource container which is bound to the exiting task cannot be freed during exit as it is needed by the scheduler. It is freed when parent calls the wait.

3.5 System Initialization

During system initialization the resource container initialization routine *rc_init()* is called. This initializes the root container. It allocates RCMAXCPU to the root container, which is the value that is used to schedule all the processes system wide in one traversal of the hierarchy. It sets the cpu reservation of the container as the maximum reservation possible. The resource container for "idle task" is set to root, but the task is not added to the scheduler bindings of the root since we do not want it to be scheduled when there is some other task in the system. The parent container of the root is set to root itself. Current running container is set to the root container.

Process 0 creates *init* process with the CLONE_RC flag set so that a new container is created for *init* task. The scheduling parameters of this new container is set so that it is entitled to all the resources root can have. *Init* process spawns other kernel threads with CLONE_RC flag set.

3.6 */proc* Interface

Linux provides, in a portable way, information on the current status of the kernel and running processes through the */proc* interface. We have changed */proc* to support resource containers.

Resource containers are represented as directories in */proc*, named *rc<resource_container_id>*. The files in this directory are a read only file *status* and a write only file *cmd*.

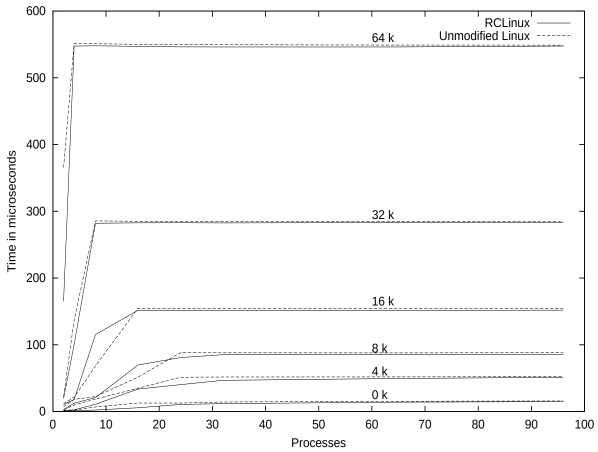

Figure 4: Context switch time for RCLinux and Unmodified Linux for processes of various sizes

We can get various attributes of a resource container like the list of child containers, the list of child processes, resource usages etc by reading the corresponding *status* file. This information can be used for debugging and system administrative purposes.

The file *cmd* is used for giving various commands to the resource container. Here the name of the API and its argument is directly written to the file. This helps in easy use of resource container functions.

3.7 Other Changes

One of the important requirements for the success of proper accounting and scheduling is the ability to keep track of time accurately. Linux uses a clock which ticks HZ times per second with HZ set as 100. This will not help us to do proper accounting of the activities since we get only a resolution of 10ms. Modern CPUs provide a timestamp counter which is incremented for every CPU clock. We have used the timestamp counter for accurately measuring the time.

Another change we have made to the Linux kernel is to the routines which insert or delete a process on the run queue. A process is inserted/deleted to/from the run-queue of the resource container to which it is bound, rather than to a global run-queue. Also whenever a process is inserted to (or deleted from) a run-queue, the *runnable* flag of the associated resource container is updated and this updating is propagated till the root.

Whenever a resource container is freed, the *last_run* array is examined and all the elements of the array pointing to the container is set to NULL.

operation	RCLinux	Linux
Simple syscall	0.71	0.71
Simple read/write	1.04	1.05
Simple stat	5.94	5.88
Simple open/close	7.66	7.54
Select on 100 tcp fd's	35.95	35.32
Signal handler installation	2.18	2.23
Signal handler overhead	2.92	2.92
Protection fault	1.48	1.48
Pipe latency	7.95	9.30
Process fork+exit	394.57	440.46
Process fork+execve	4109.00	4139.00
Process fork+/bin/sh	15023.00	14977.00

Table 1: Time taken for various operations(in micro seconds)

4 Experimental Evaluation

4.1 Performance overhead

We ran *lmbench*[13] to evaluate the performance of RCLinux compared to an unmodified version of Linux. The tests were performed on a 400 MHz Celeron with 128MB memory. We found that the overhead was very minimal. The results are summerized in table 1.

Figure 4 shows the context switch time for different number of processes for different sizes. The processes are connected in a ring of Unix pipes. Each process reads a token from its pipe, possibly does some work, and then writes the token to the next process. A size of zero is the baseline process that does nothing except pass the token on to the next process. A process size of greater than zero means that the process does some work before passing on the token. The work is simulated as the summing up of an array of the specified size. The graph shows that RCLinux performs slightly better when number of processes is large.

4.2 Use of fixed CPU share

In this experiment, we create three child containers C1, C2 and C3 for the resource container to which *init* is bound, having a fixed CPU share of 16%, 32% and 48% respectively. We also create three CPU bound jobs R1, R2 and R3. The cumulative execution time of these jobs are shown in figure 5.

The figure shows that when all the processes are running, R1, R2 and R3 get 14.3%, 28.6% and 42.9% of the CPU respectively. There is a proportional decrease in the CPU allotted than what has been specified and may be due to the lower resolution of the CPU timer[4], pre-emption being done only at safe points and the non-accounting of some of the activities in the system (eg. network activities that are

[4]Eventhough we use CPU time stamp counter for keeping track of time, we get timer interrupts only in 10ms intervals.

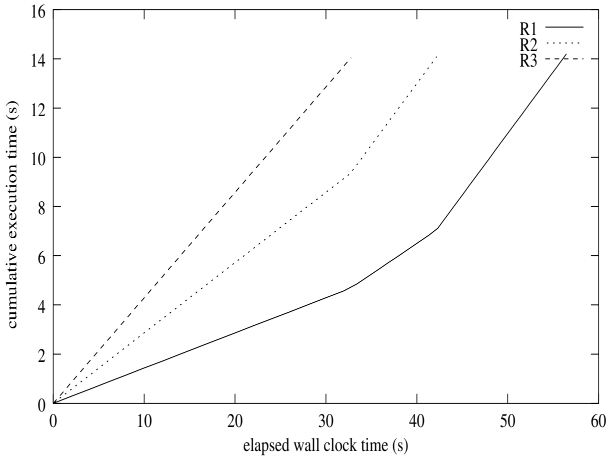

Figure 5: Proportional allocation of CPU for fixed share containers

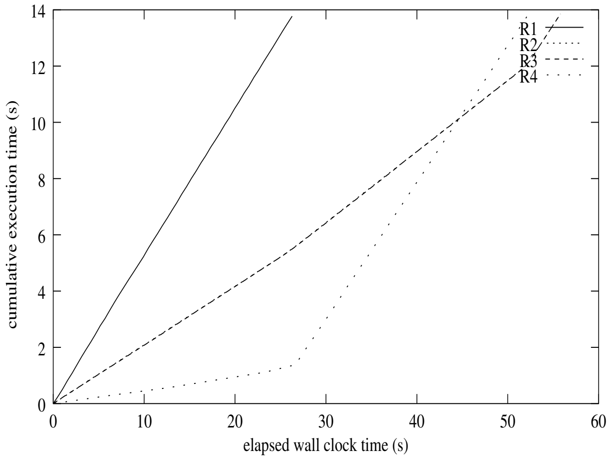

Figure 6: Proportional allocation of CPU for fixed share and shared containers

charged to *rc_system*, the time taken for scheduling etc.). Once R3 exits, R1 and R2 get 25.2% and 50.1% of the CPU respectively and the remaining CPU power is used by various system activities and other processes in the system. This is more than what has been specified as there are not many other activities in the system. Once R2 also exits, R1 gets 50.0% of the CPU.

4.3 Allocation of CPU in presence of fixed share and shared containers

Another experiment was conducted in which three fixed share containers and a shared container were created. Fixed containers had a CPU share of 48%, 16% and 16%. CPU bound jobs R1, R2 and R3 were bound to these containers. Another CPU bound job R4 was bound to a shared container. All the four jobs were same. The cumulative execution time of these containers are shown in figure 6.

When all the processes were running, R1, R2, R3 and R4 got 54.2%, 20.9%, 20.9% and 8.7% of CPU time respectively. When R1 exited, the shared container got most of the CPU that R1 was using. R2 and R3 got 25.2% and R4 got 48.9% of the CPU. R4 completed before R2 and R4. When R4 exited, R2 and R3 got 49.5% of the CPU.

4.4 Scheduling for Multimedia

Multimedia applications often perform poorly under Linux when there are some CPU bound processes running. For this experiment, we have run a multimedia application in the presence of CPU bound processes - each being a distributed genetic algorithm client[5]. *mpeg_play* was used for viewing the mpeg file. Frame rate was noted for different number of genetic algorithm clients. The frame rate observed under Linux was 15.5, 9.8, 7.0 and 5.6 for 1, 3, 5 and 7 clients respectively.

Even if the frame rate was 15.5 with one client, the movie was jerky. This was because of the way Linux schedules processes. Initially both *mpeg_play* and the client will be allocated 200ms (ie *count* = 20) CPU time. Once *mpeg_play* is scheduled, it will require service from the X server and it will sleep. Next the client is scheduled, and it will use up its 200ms since it does not sleep in between. Again, mpeg_play is scheduled and it will again sleep for service from X server. So the X server and *mpeg_play* will be scheduled alternately till their remaining CPU quanta is exhausted (ie *count* becomes 0). After this, the whole cycle repeats. The movie is jerky as the client runs once per this major cycle.

Next, we ran *mpeg_play* by creating two resource containers with fixed cpu share and binding the *mpeg_play* and X server processes to those containers. We tried various amounts of CPU shares from 10% to 60%. We also allocated 30% to 50% of the CPU to X server. But the maximum frame rate went only up to 7.0.

The problem in this case was that even if the scheduler was allocating enough CPU time to both *mpeg_play* and X server, they were not able to use it since one needed the service of the other. After *mpeg_play* generated a frame, it required the service of the X server to display it. After displaying the frame, the X server needed *mpeg_play* to be scheduled to generate the next frame.

Now we ran *mpeg_play* by creating a single resource container with fixed cpu share and binding both *mpeg_play* and X server processes to that container. We tested it by allocating different amount of CPUs to this resource container and running distributed genetic algorithm with different number of clients.

With this arrangement for binding processes, we obtained significant performance for CPU shares more than 60%. Also the picture was very smooth (no jerks) even for

[5]This application recognizes digits using a number of computational nodes (clients) with the server using many clients. Most of the time server sleeps. We have run all the clients on the same machine.

No. of clients	CPU share								
	10%	20%	30%	40%	50%	60%	70%	80%	90%
1	4.7	5.2	10.0	10.4	15.1	15.5	19.9	20.7	25.3
3	4.7	5.2	9.9	10.4	15.0	15.6	19.9	20.7	25.3
5	4.7	5.2	10.0	10.3	15.0	15.6	19.9	20.7	25.3

Table 2: Frame rate using resource containers

lower frame rates. The resulting framerates are shown in Table 1. The output shows that the frame rate is independent of number of clients running in the system and depends only on the CPU allocated to the resource container.

5 Conclusions and Future Work

The concept of resource container was introduced to address the lack of appropriate support for server applications in existing operating systems. We have extended it by introducing multiple scheduling classes and by using container specific schedulers. We have shown that we can use resource container mechanism for predictable performance of applications and for scheduling multimedia applications.

We have implemented resource containers for Linux by making minimal changes to the existing kernel which is designed under the assumption that a process is a resource principal. We have modified only two kernel data structures, *task_struct* and *file*. We have changed a total of 12 .c files, 3 .h files and 1 .S file in the kernel, fs, net, init and arch directories.

The statistics information that is maintained for disk accesses and network packets can be used for taking better CPU and I/O scheduling decisions. A detailed study has to be conducted to find out the influence these have on the priority computations.

We have not looked into memory management subsystem in this resource container framework. When a group of processes are performing related activities, the working set of each process may depend on the working set of the other processes in the group. So better paging decisions may be possible if resource container framework is added to the memory management subsystem. We have also not looked into multiprocessor related issues in our design.

Acknowledgements: We thank N.Ganesh for his help in running lmbench on RCLinux and Linux-2.2.5 and Stephen Tweedie, our shepherd, for his many suggestions on improving the paper. Financial support from Veritas Software, Pune is also gratefully acknowledged.

References

[1] Gaurav Banga, Peter Druschel, and Jeffery C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of Third Symp. on OS Design and Implementation*, New Orleans, Feb 1999.

[2] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Prentice Hall of Australia Pty Ltd, 1994.

[3] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX schedular unacceptable for multimedia applications. In *Proc. of Fourth International Workshop on Network and OS Support for Digital Audio and Video*, Nov 1993.

[4] Jason Nieh and Monica S. Lam. The design of SMART: A scheduler for multimedia applications. CSL-TR-96-697, Stanford University, Jun 1996.

[5] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proc. of Second Symp. on OS Design and Implementation*, Seattle, WA, Oct 1996.

[6] Gaurav Banga. *Operating System Support fo Server Applications*. PhD thesis, Rice University, May 1999.

[7] Resource containers and LRP for FreeBSD. http://www.cs.rice.edu/CS/Systems/ScalaServer/code/rescon-lrp/README.html.

[8] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proc. of Second Symp. on OS Design and Implementation*, Seattle, WA, Oct 1996.

[9] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shui Oikawa. Resource kernels: A resource-centric approach to RT systems. In *Proc. of SPIE/ACM Conf. on Multimedia Computing & Networking*, Jan 1998.

[10] Shui Oikawa and Raj Rajkumar. Linux/RK: A portable resource kernel in Linux. In *IEEE Real-Time Systems Symp. Work-In-Progress*, Dec 1998.

[11] RTLinux. http://luz.cs.nmt.edu/ rtlinux.

[12] KURT: The KU Real-Time Linux. http://hegel.ittc.ukans.edu/projects/kurt.

[13] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proc. of 1996 USENIX Technical Conf.*, San Diego, CA, Jan 1996.