CPUSETS

Written by Simon.Derr@bull.net

-
- Modified by Paul Jackson <pj@sgi.com>
- Modified by Christoph Lameter <cl@linux.com>
- Modified by Paul Menage <menage@google.com>
- Modified by Hidetoshi Seto <seto.hidetoshi@jp.fujitsu.com>

1. Cpusets

1.1 What are cpusets ?

Cpusets provide a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks. In this document "Memory Node" refers to an on-line node that contains memory.

Cpusets constrain the CPU and Memory placement of tasks to only the resources within a task's current cpuset. They form a nested hierarchy visible in a virtual file system. These are the essential hooks, beyond what is already present, required to manage dynamic job placement on large systems.

Cpusets use the generic cgroup subsystem described in Documentation/admin-guide/cgroup-v1/cgroups.rst.

Requests by a task, using the sched_setaffinity(2) system call to include CPUs in its CPU affinity mask, and using the mbind(2) and set_mempolicy(2) system calls to include Memory Nodes in its memory policy, are both filtered through that task's cpuset, filtering out any CPUs or Memory Nodes not in that cpuset. The scheduler will not schedule a task on a CPU that is not allowed in its cpus_allowed vector, and the kernel page allocator will not allocate a page on a node that is not allowed in the requesting task's mems_allowed vector.

User level code may create and destroy cpusets by name in the cgroup virtual file system, manage the attributes and permissions of these cpusets and which CPUs and Memory Nodes are assigned to each cpuset, specify and query to which cpuset a task is assigned, and list the task pids assigned to a cpuset.

1.2 Why are cpusets needed ?

The management of large computer systems, with many processors (CPUs), complex memory cache hierarchies and multiple Memory Nodes having non-uniform access times (NUMA) presents additional challenges for the efficient scheduling and memory placement of processes.

Frequently more modest sized systems can be operated with adequate efficiency just by letting the operating system automatically share the available CPU and Memory resources amongst the requesting tasks.

But larger systems, which benefit more from careful processor and memory placement to reduce memory access times and contention, and which typically represent a larger investment for the customer, can benefit from explicitly placing jobs on properly sized subsets of the system.

This can be especially valuable on:

- Web Servers running multiple instances of the same web application,
- Servers running different applications (for instance, a web server and a database), or
- NUMA systems running large HPC applications with demanding performance characteristics.

These subsets, or "soft partitions" must be able to be dynamically adjusted, as the job mix changes, without impacting other concurrently executing jobs. The location of the running jobs pages may also be moved when the memory locations are changed.

The kernel cpuset patch provides the minimum essential kernel mechanisms required to efficiently implement such subsets. It leverages existing CPU and Memory Placement facilities in the Linux kernel to avoid any additional impact on the critical scheduler or memory allocator code.

1.3 How are cpusets implemented ?

Cpusets provide a Linux kernel mechanism to constrain which CPUs and Memory Nodes are used by a process or set of processes.

The Linux kernel already has a pair of mechanisms to specify on which CPUs a task may be scheduled (sched_setaffinity) and on which Memory Nodes it may obtain memory (mbind, set_mempolicy).

Cpusets extends these two mechanisms as follows:

- Cpusets are sets of allowed CPUs and Memory Nodes, known to the kernel.
- Each task in the system is attached to a cpuset, via a pointer in the task structure to a reference counted cgroup

structure.
- Calls to sched_setaffinity are filtered to just those CPUs allowed in that task's cpuset.
- Calls to mbind and set_mempolicy are filtered to just those Memory Nodes allowed in that task's cpuset.
- The root cpuset contains all the systems CPUs and Memory Nodes.
- For any cpuset, one can define child cpusets containing a subset of the parents CPU and Memory Node resources.
- The hierarchy of cpusets can be mounted at /dev/cpuset, for browsing and manipulation from user space.
- A cpuset may be marked exclusive, which ensures that no other cpuset (except direct ancestors and descendants) may contain any overlapping CPUs or Memory Nodes.
- You can list all the tasks (by pid) attached to any cpuset.

The implementation of cpusets requires a few, simple hooks into the rest of the kernel, none in performance critical paths:

- in init/main.c, to initialize the root cpuset at system boot.
- in fork and exit, to attach and detach a task from its cpuset.
- in sched_setaffinity, to mask the requested CPUs by what's allowed in that task's cpuset.
- in sched.c migrate_live_tasks(), to keep migrating tasks within the CPUs allowed by their cpuset, if possible.
- in the mbind and set_mempolicy system calls, to mask the requested Memory Nodes by what's allowed in that task's cpuset.
- in page_alloc.c, to restrict memory to allowed nodes.
- in vmscan.c, to restrict page recovery to the current cpuset.

You should mount the "cgroup" filesystem type in order to enable browsing and modifying the cpusets presently known to the kernel. No new system calls are added for cpusets - all support for querying and modifying cpusets is via this cpuset file system.

The /proc/<pid>/status file for each task has four added lines, displaying the task's cpus_allowed (on which CPUs it may be scheduled) and mems_allowed (on which Memory Nodes it may obtain memory), in the two formats seen in the following example:

```
Cpus_allowed:   ffffffff,ffffffff,ffffffff,ffffffff
Cpus_allowed_list:      0-127
Mems_allowed:   ffffffff,ffffffff
Mems_allowed_list:      0-63
```

Each cpuset is represented by a directory in the cgroup file system containing (on top of the standard cgroup files) the following files describing that cpuset:

- cpuset.cpus: list of CPUs in that cpuset
- cpuset.mems: list of Memory Nodes in that cpuset
- cpuset.memory_migrate flag: if set, move pages to cpusets nodes
- cpuset.cpu_exclusive flag: is cpu placement exclusive?
- cpuset.mem_exclusive flag: is memory placement exclusive?
- cpuset.mem_hardwall flag: is memory allocation hardwalled
- cpuset.memory_pressure: measure of how much paging pressure in cpuset
- cpuset.memory_spread_page flag: if set, spread page cache evenly on allowed nodes
- cpuset.memory_spread_slab flag: if set, spread slab cache evenly on allowed nodes
- cpuset.sched_load_balance flag: if set, load balance within CPUs on that cpuset
- cpuset.sched_relax_domain_level: the searching range when migrating tasks

In addition, only the root cpuset has the following file:

- cpuset.memory_pressure_enabled flag: compute memory_pressure?

New cpusets are created using the mkdir system call or shell command. The properties of a cpuset, such as its flags, allowed CPUs and Memory Nodes, and attached tasks, are modified by writing to the appropriate file in that cpusets directory, as listed above.

The named hierarchical structure of nested cpusets allows partitioning a large system into nested, dynamically changeable, "soft-partitions".

The attachment of each task, automatically inherited at fork by any children of that task, to a cpuset allows organizing the work load on a system into related sets of tasks such that each set is constrained to using the CPUs and Memory Nodes of a particular cpuset. A task may be re-attached to any other cpuset, if allowed by the permissions on the necessary cpuset file system directories.

Such management of a system "in the large" integrates smoothly with the detailed placement done on individual tasks and memory regions using the sched_setaffinity, mbind and set_mempolicy system calls.

The following rules apply to each cpuset:

- Its CPUs and Memory Nodes must be a subset of its parents.
- It can't be marked exclusive unless its parent is.
- If its cpu or memory is exclusive, they may not overlap any sibling.

These rules, and the natural hierarchy of cpusets, enable efficient enforcement of the exclusive guarantee, without having to scan all cpusets every time any of them change to ensure nothing overlaps a exclusive cpuset. Also, the use of a Linux virtual file system (vfs) to represent the cpuset hierarchy provides for a familiar permission and name space for cpusets, with a minimum of additional kernel code.

The cpus and mems files in the root (top_cpuset) cpuset are read-only. The cpus file automatically tracks the value of cpu_online_mask using a CPU hotplug notifier, and the mems file automatically tracks the value of node_states[N_MEMORY]--i.e., nodes with memory--using the cpuset_track_online_nodes() hook.

The cpuset.effective_cpus and cpuset.effective_mems files are normally read-only copies of cpuset.cpus and cpuset.mems files respectively. If the cpuset cgroup filesystem is mounted with the special "cpuset_v2_mode" option, the behavior of these files will become similar to the corresponding files in cpuset v2. In other words, hotplug events will not change cpuset.cpus and cpuset.mems. Those events will only affect cpuset.effective_cpus and cpuset.effective_mems which show the actual cpus and memory nodes that are currently used by this cpuset. See Documentation/admin-guide/cgroup-v2.rst for more information about cpuset v2 behavior.

1.4 What are exclusive cpusets ?

If a cpuset is cpu or mem exclusive, no other cpuset, other than a direct ancestor or descendant, may share any of the same CPUs or Memory Nodes.

A cpuset that is cpuset.mem_exclusive *or* cpuset.mem_hardwall is "hardwalled", i.e. it restricts kernel allocations for page, buffer and other data commonly shared by the kernel across multiple users. All cpusets, whether hardwalled or not, restrict allocations of memory for user space. This enables configuring a system so that several independent jobs can share common kernel data, such as file system pages, while isolating each job's user allocation in its own cpuset. To do this, construct a large mem_exclusive cpuset to hold all the jobs, and construct child, non-mem_exclusive cpusets for each individual job. Only a small amount of typical kernel memory, such as requests from interrupt handlers, is allowed to be taken outside even a mem_exclusive cpuset.

1.5 What is memory_pressure ?

The memory_pressure of a cpuset provides a simple per-cpuset metric of the rate that the tasks in a cpuset are attempting to free up in use memory on the nodes of the cpuset to satisfy additional memory requests.

This enables batch managers monitoring jobs running in dedicated cpusets to efficiently detect what level of memory pressure that job is causing.

This is useful both on tightly managed systems running a wide mix of submitted jobs, which may choose to terminate or re-prioritize jobs that are trying to use more memory than allowed on the nodes assigned to them, and with tightly coupled, long running, massively parallel scientific computing jobs that will dramatically fail to meet required performance goals if they start to use more memory than allowed to them.

This mechanism provides a very economical way for the batch manager to monitor a cpuset for signs of memory pressure. It's up to the batch manager or other user code to decide what to do about it and take action.

==>

> Unless this feature is enabled by writing "1" to the special file /dev/cpuset/memory_pressure_enabled, the hook in the rebalance code of __alloc_pages() for this metric reduces to simply noticing that the cpuset_memory_pressure_enabled flag is zero. So only systems that enable this feature will compute the metric.

Why a per-cpuset, running average:

> Because this meter is per-cpuset, rather than per-task or mm, the system load imposed by a batch scheduler monitoring this metric is sharply reduced on large systems, because a scan of the tasklist can be avoided on each set of queries.
>
> Because this meter is a running average, instead of an accumulating counter, a batch scheduler can detect memory pressure with a single read, instead of having to read and accumulate results for a period of time.
>
> Because this meter is per-cpuset rather than per-task or mm, the batch scheduler can obtain the key information, memory pressure in a cpuset, with a single read, rather than having to query and accumulate results over all the (dynamically changing) set of tasks in the cpuset.

A per-cpuset simple digital filter (requires a spinlock and 3 words of data per-cpuset) is kept, and updated by any task attached to that cpuset, if it enters the synchronous (direct) page reclaim code.

A per-cpuset file provides an integer number representing the recent (half-life of 10 seconds) rate of direct page reclaims caused by the tasks in the cpuset, in units of reclaims attempted per second, times 1000.

1.6 What is memory spread ?

There are two boolean flag files per cpuset that control where the kernel allocates pages for the file system buffers and related in kernel data structures. They are called 'cpuset.memory_spread_page' and 'cpuset.memory_spread_slab'.

If the per-cpuset boolean flag file 'cpuset.memory_spread_page' is set, then the kernel will spread the file system buffers (page cache) evenly over all the nodes that the faulting task is allowed to use, instead of preferring to put those pages on the node where the task is running.

If the per-cpuset boolean flag file 'cpuset.memory_spread_slab' is set, then the kernel will spread some file system related slab caches, such as for inodes and dentries evenly over all the nodes that the faulting task is allowed to use, instead of preferring to put those pages on the node where the task is running.

The setting of these flags does not affect anonymous data segment or stack segment pages of a task.

By default, both kinds of memory spreading are off, and memory pages are allocated on the node local to where the task is running, except perhaps as modified by the task's NUMA mempolicy or cpuset configuration, so long as sufficient free memory pages are available.

When new cpusets are created, they inherit the memory spread settings of their parent.

Setting memory spreading causes allocations for the affected page or slab caches to ignore the task's NUMA mempolicy and be spread instead. Tasks using mbind() or set_mempolicy() calls to set NUMA mempolicies will not notice any change in these calls as a result of their containing task's memory spread settings. If memory spreading is turned off, then the currently specified NUMA mempolicy once again applies to memory page allocations.

Both 'cpuset.memory_spread_page' and 'cpuset.memory_spread_slab' are boolean flag files. By default they contain "0", meaning that the feature is off for that cpuset. If a "1" is written to that file, then that turns the named feature on.

The implementation is simple.

Setting the flag 'cpuset.memory_spread_page' turns on a per-process flag PFA_SPREAD_PAGE for each task that is in that cpuset or subsequently joins that cpuset. The page allocation calls for the page cache is modified to perform an inline check for this PFA_SPREAD_PAGE task flag, and if set, a call to a new routine cpuset_mem_spread_node() returns the node to prefer for the allocation.

Similarly, setting 'cpuset.memory_spread_slab' turns on the flag PFA_SPREAD_SLAB, and appropriately marked slab caches will allocate pages from the node returned by cpuset_mem_spread_node().

The cpuset_mem_spread_node() routine is also simple. It uses the value of a per-task rotor cpuset_mem_spread_rotor to select the next node in the current task's mems_allowed to prefer for the allocation.

This memory placement policy is also known (in other contexts) as round-robin or interleave.

This policy can provide substantial improvements for jobs that need to place thread local data on the corresponding node, but that need to access large file system data sets that need to be spread across the several nodes in the jobs cpuset in order to fit. Without this policy, especially for jobs that might have one thread reading in the data set, the memory allocation across the nodes in the jobs cpuset can become very uneven.

1.7 What is sched_load_balance ?

The kernel scheduler (kernel/sched/core.c) automatically load balances tasks. If one CPU is underutilized, kernel code running on that CPU will look for tasks on other more overloaded CPUs and move those tasks to itself, within the constraints of such placement mechanisms as cpusets and sched_setaffinity.

The algorithmic cost of load balancing and its impact on key shared kernel data structures such as the task list increases more than linearly with the number of CPUs being balanced. So the scheduler has support to partition the systems CPUs into a number of sched domains such that it only load balances within each sched domain. Each sched domain covers some subset of the CPUs in the system; no two sched domains overlap; some CPUs might not be in any sched domain and hence won't be load balanced.

Put simply, it costs less to balance between two smaller sched domains than one big one, but doing so means that overloads in one of the two domains won't be load balanced to the other one.

By default, there is one sched domain covering all CPUs, including those marked isolated using the kernel boot time "isolcpus=" argument. However, the isolated CPUs will not participate in load balancing, and will not have tasks running on them unless explicitly assigned.

This default load balancing across all CPUs is not well suited for the following two situations:

1. On large systems, load balancing across many CPUs is expensive. If the system is managed using cpusets to place independent jobs on separate sets of CPUs, full load balancing is unnecessary.
2. Systems supporting realtime on some CPUs need to minimize system overhead on those CPUs, including avoiding task load balancing if that is not needed.

When the per-cpuset flag "cpuset.sched_load_balance" is enabled (the default setting), it requests that all the CPUs in that cpusets allowed 'cpuset.cpus' be contained in a single sched domain, ensuring that load balancing can move a task (not otherwised pinned, as by sched_setaffinity) from any CPU in that cpuset to any other.

When the per-cpuset flag "cpuset.sched_load_balance" is disabled, then the scheduler will avoid load balancing across the CPUs in that cpuset, --except-- in so far as is necessary because some overlapping cpuset has "sched_load_balance" enabled.

So, for example, if the top cpuset has the flag "cpuset.sched_load_balance" enabled, then the scheduler will have one sched domain covering all CPUs, and the setting of the "cpuset.sched_load_balance" flag in any other cpusets won't matter, as we're already fully load balancing.

Therefore in the above two situations, the top cpuset flag "cpuset.sched_load_balance" should be disabled, and only some of the smaller, child cpusets have this flag enabled.

When doing this, you don't usually want to leave any unpinned tasks in the top cpuset that might use non-trivial amounts of CPU, as such tasks may be artificially constrained to some subset of CPUs, depending on the particulars of this flag setting in descendant cpusets. Even if such a task could use spare CPU cycles in some other CPUs, the kernel scheduler might not consider the possibility of load balancing that task to that underused CPU.

Of course, tasks pinned to a particular CPU can be left in a cpuset that disables "cpuset.sched_load_balance" as those tasks aren't going anywhere else anyway.

There is an impedance mismatch here, between cpusets and sched domains. Cpusets are hierarchical and nest. Sched domains are flat; they don't overlap and each CPU is in at most one sched domain.

It is necessary for sched domains to be flat because load balancing across partially overlapping sets of CPUs would risk unstable dynamics that would be beyond our understanding. So if each of two partially overlapping cpusets enables the flag 'cpuset.sched_load_balance', then we form a single sched domain that is a superset of both. We won't move a task to a CPU outside its cpuset, but the scheduler load balancing code might waste some compute cycles considering that possibility.

This mismatch is why there is not a simple one-to-one relation between which cpusets have the flag "cpuset.sched_load_balance" enabled, and the sched domain configuration. If a cpuset enables the flag, it will get balancing across all its CPUs, but if it disables the flag, it will only be assured of no load balancing if no other overlapping cpuset enables the flag.

If two cpusets have partially overlapping 'cpuset.cpus' allowed, and only one of them has this flag enabled, then the other may find its tasks only partially load balanced, just on the overlapping CPUs. This is just the general case of the top_cpuset example given a few paragraphs above. In the general case, as in the top cpuset case, don't leave tasks that might use non-trivial amounts of CPU in such partially load balanced cpusets, as they may be artificially constrained to some subset of the CPUs allowed to them, for lack of load balancing to the other CPUs.

CPUs in "cpuset.isolcpus" were excluded from load balancing by the isolcpus= kernel boot option, and will never be load balanced regardless of the value of "cpuset.sched_load_balance" in any cpuset.

1.7.1 sched_load_balance implementation details.

The per-cpuset flag 'cpuset.sched_load_balance' defaults to enabled (contrary to most cpuset flags.) When enabled for a cpuset, the kernel will ensure that it can load balance across all the CPUs in that cpuset (makes sure that all the CPUs in the cpus_allowed of that cpuset are in the same sched domain.)

If two overlapping cpusets both have 'cpuset.sched_load_balance' enabled, then they will be (must be) both in the same sched domain.

If, as is the default, the top cpuset has 'cpuset.sched_load_balance' enabled, then by the above that means there is a single sched domain covering the whole system, regardless of any other cpuset settings.

The kernel commits to user space that it will avoid load balancing where it can. It will pick as fine a granularity partition of sched domains as it can while still providing load balancing for any set of CPUs allowed to a cpuset having 'cpuset.sched_load_balance' enabled.

The internal kernel cpuset to scheduler interface passes from the cpuset code to the scheduler code a partition of the load balanced CPUs in the system. This partition is a set of subsets (represented as an array of struct cpumask) of CPUs, pairwise disjoint, that cover all the CPUs that must be load balanced.

The cpuset code builds a new such partition and passes it to the scheduler sched domain setup code, to have the sched domains rebuilt as necessary, whenever:

- the 'cpuset.sched_load_balance' flag of a cpuset with non-empty CPUs changes,
- or CPUs come or go from a cpuset with this flag enabled,
- or 'cpuset.sched_relax_domain_level' value of a cpuset with non-empty CPUs and with this flag enabled changes,
- or a cpuset with non-empty CPUs and with this flag enabled is removed,
- or a cpu is offlined/onlined.

This partition exactly defines what sched domains the scheduler should setup - one sched domain for each element (struct cpumask) in the partition.

The scheduler remembers the currently active sched domain partitions. When the scheduler routine partition_sched_domains() is invoked from the cpuset code to update these sched domains, it compares the new partition requested with the current, and updates its sched domains, removing the old and adding the new, for each change.

1.8 What is sched_relax_domain_level ?

In sched domain, the scheduler migrates tasks in 2 ways; periodic load balance on tick, and at time of some schedule events.

When a task is woken up, scheduler try to move the task on idle CPU. For example, if a task A running on CPU X activates another task B on the same CPU X, and if CPU Y is X's sibling and performing idle, then scheduler migrate task B to CPU Y so that task B can start on CPU Y without waiting task A on CPU X.

And if a CPU run out of tasks in its runqueue, the CPU try to pull extra tasks from other busy CPUs to help them before it is going to be idle.

Of course it takes some searching cost to find movable tasks and/or idle CPUs, the scheduler might not search all CPUs in the domain every time. In fact, in some architectures, the searching ranges on events are limited in the same socket or node where the CPU locates, while the load balance on tick searches all.

For example, assume CPU Z is relatively far from CPU X. Even if CPU Z is idle while CPU X and the siblings are busy, scheduler can't migrate woken task B from X to Z since it is out of its searching range. As the result, task B on CPU X need to wait task A or wait load balance on the next tick. For some applications in special situation, waiting 1 tick may be too long.

The 'cpuset.sched_relax_domain_level' file allows you to request changing this searching range as you like. This file takes int value which indicates size of searching range in levels ideally as follows, otherwise initial value -1 that indicates the cpuset has no request.

-1	no request. use system default or follow request of others.
0	no search.
1	search siblings (hyperthreads in a core).
2	search cores in a package.
3	search cpus in a node [= system wide on non-NUMA system]
4	search nodes in a chunk of node [on NUMA system]
5	search system wide [on NUMA system]

The system default is architecture dependent. The system default can be changed using the relax_domain_level= boot parameter.

This file is per-cpuset and affect the sched domain where the cpuset belongs to. Therefore if the flag 'cpuset.sched_load_balance' of a cpuset is disabled, then 'cpuset.sched_relax_domain_level' have no effect since there is no sched domain belonging the cpuset.

If multiple cpusets are overlapping and hence they form a single sched domain, the largest value among those is used. Be careful, if one requests 0 and others are -1 then 0 is used.

Note that modifying this file will have both good and bad effects, and whether it is acceptable or not depends on your situation. Don't modify this file if you are not sure.

If your situation is:

- The migration costs between each cpu can be assumed considerably small(for you) due to your special application's behavior or special hardware support for CPU cache etc.
- The searching cost doesn't have impact(for you) or you can make the searching cost enough small by managing cpuset to compact etc.
- The latency is required even it sacrifices cache hit rate etc. then increasing 'sched_relax_domain_level' would benefit you.

1.9 How do I use cpusets ?

In order to minimize the impact of cpusets on critical kernel code, such as the scheduler, and due to the fact that the kernel does not support one task updating the memory placement of another task directly, the impact on a task of changing its cpuset CPU or Memory Node placement, or of changing to which cpuset a task is attached, is subtle.

If a cpuset has its Memory Nodes modified, then for each task attached to that cpuset, the next time that the kernel attempts to allocate a page of memory for that task, the kernel will notice the change in the task's cpuset, and update its per-task memory placement to remain within the new cpusets memory placement. If the task was using mempolicy MPOL_BIND, and the nodes to which it was bound overlap with its new cpuset, then the task will continue to use whatever subset of MPOL_BIND nodes are still allowed in the new cpuset. If the task was using MPOL_BIND and now none of its MPOL_BIND nodes are allowed in the new cpuset, then the task will be essentially treated as if it was MPOL_BIND bound to the new cpuset (even though its NUMA placement, as queried by get_mempolicy(), doesn't change). If a task is moved from one cpuset to another, then the kernel will adjust the task's memory placement, as above, the next time that the kernel attempts to allocate a page of memory for that task.

If a cpuset has its 'cpuset.cpus' modified, then each task in that cpuset will have its allowed CPU placement changed immediately. Similarly, if a task's pid is written to another cpuset's 'tasks' file, then its allowed CPU placement is changed immediately. If such a task had been bound to some subset of its cpuset using the sched_setaffinity() call, the task will be allowed to run on any CPU allowed in its new cpuset, negating the effect of the prior sched_setaffinity() call.

In summary, the memory placement of a task whose cpuset is changed is updated by the kernel, on the next allocation of a page for that task, and the processor placement is updated immediately.

Normally, once a page is allocated (given a physical page of main memory) then that page stays on whatever node it was allocated, so long as it remains allocated, even if the cpusets memory placement policy 'cpuset.mems' subsequently changes. If the cpuset flag file 'cpuset.memory_migrate' is set true, then when tasks are attached to that cpuset, any pages that task had allocated to it on nodes in its previous cpuset are migrated to the task's new cpuset. The relative placement of the page within the cpuset is preserved during these migration operations if possible. For example if the page was on the second valid node of the prior cpuset then the page will be placed on the second valid node of the new cpuset.

Also if 'cpuset.memory_migrate' is set true, then if that cpuset's 'cpuset.mems' file is modified, pages allocated to tasks in that cpuset, that were on nodes in the previous setting of 'cpuset.mems', will be moved to nodes in the new setting of 'mems.' Pages that were not in the task's prior cpuset, or in the cpuset's prior 'cpuset.mems' setting, will not be moved.

There is an exception to the above. If hotplug functionality is used to remove all the CPUs that are currently assigned to a cpuset, then all the tasks in that cpuset will be moved to the nearest ancestor with non-empty cpus. But the moving of some (or all) tasks might fail if cpuset is bound with another cgroup subsystem which has some restrictions on task attaching. In this failing case, those tasks will stay in the original cpuset, and the kernel will automatically update their cpus_allowed to allow all online CPUs. When memory hotplug functionality for removing Memory Nodes is available, a similar exception is expected to apply there as well. In general, the kernel prefers to violate cpuset placement, over starving a task that has had all its allowed CPUs or Memory Nodes taken offline.

There is a second exception to the above. GFP_ATOMIC requests are kernel internal allocations that must be satisfied, immediately. The kernel may drop some request, in rare cases even panic, if a GFP_ATOMIC alloc fails. If the request cannot be satisfied within the current task's cpuset, then we relax the cpuset, and look for memory anywhere we can find it. It's better to violate the cpuset than stress the kernel.

To start a new job that is to be contained within a cpuset, the steps are:

1. mkdir /sys/fs/cgroup/cpuset
2. mount -t cgroup -ocpuset cpuset /sys/fs/cgroup/cpuset
3. Create the new cpuset by doing mkdir's and write's (or echo's) in the /sys/fs/cgroup/cpuset virtual file system.
4. Start a task that will be the "founding father" of the new job.
5. Attach that task to the new cpuset by writing its pid to the /sys/fs/cgroup/cpuset tasks file for that cpuset.
6. fork, exec or clone the job tasks from this founding father task.

For example, the following sequence of commands will setup a cpuset named "Charlie", containing just CPUs 2 and 3, and Memory Node 1, and then start a subshell 'sh' in that cpuset:

```
mount -t cgroup -ocpuset cpuset /sys/fs/cgroup/cpuset
cd /sys/fs/cgroup/cpuset
mkdir Charlie
cd Charlie
/bin/echo 2-3 > cpuset.cpus
/bin/echo 1 > cpuset.mems
/bin/echo $$ > tasks
sh
# The subshell 'sh' is now running in cpuset Charlie
# The next line should display '/Charlie'
cat /proc/self/cpuset
```

There are ways to query or modify cpusets:

- via the cpuset file system directly, using the various cd, mkdir, echo, cat, rmdir commands from the shell, or their equivalent from C.
- via the C library libcpuset.
- via the C library libcgroup. (http://sourceforge.net/projects/libcg/)
- via the python application cset. (http://code.google.com/p/cpuset/)

The sched_setaffinity calls can also be done at the shell prompt using SGI's runon or Robert Love's taskset. The mbind and set_mempolicy calls can be done at the shell prompt using the numactl command (part of Andi Kleen's numa package).

2. Usage Examples and Syntax

2.1 Basic Usage

Creating, modifying, using the cpusets can be done through the cpuset virtual filesystem.

To mount it, type: # mount -t cgroup -o cpuset cpuset /sys/fs/cgroup/cpuset

Then under /sys/fs/cgroup/cpuset you can find a tree that corresponds to the tree of the cpusets in the system. For instance, /sys/fs/cgroup/cpuset is the cpuset that holds the whole system.

If you want to create a new cpuset under /sys/fs/cgroup/cpuset:

```
# cd /sys/fs/cgroup/cpuset
# mkdir my_cpuset
```

Now you want to do something with this cpuset:

```
# cd my_cpuset
```

In this directory you can find several files:

```
# ls
cgroup.clone_children  cpuset.memory_pressure
cgroup.event_control   cpuset.memory_spread_page
cgroup.procs           cpuset.memory_spread_slab
cpuset.cpu_exclusive   cpuset.mems
cpuset.cpus            cpuset.sched_load_balance
cpuset.mem_exclusive   cpuset.sched_relax_domain_level
cpuset.mem_hardwall    notify_on_release
cpuset.memory_migrate  tasks
```

Reading them will give you information about the state of this cpuset: the CPUs and Memory Nodes it can use, the processes that are using it, its properties. By writing to these files you can manipulate the cpuset.

Set some flags:

```
# /bin/echo 1 > cpuset.cpu_exclusive
```

Add some cpus:

```
# /bin/echo 0-7 > cpuset.cpus
```

Add some mems:

```
# /bin/echo 0-7 > cpuset.mems
```

Now attach your shell to this cpuset:

```
# /bin/echo $$ > tasks
```

You can also create cpusets inside your cpuset by using mkdir in this directory:

```
# mkdir my_sub_cs
```

To remove a cpuset, just use rmdir:

```
# rmdir my_sub_cs
```

This will fail if the cpuset is in use (has cpusets inside, or has processes attached).

Note that for legacy reasons, the "cpuset" filesystem exists as a wrapper around the cgroup filesystem.

The command:

```
mount -t cpuset X /sys/fs/cgroup/cpuset
```

is equivalent to:

```
mount -t cgroup -ocpuset,noprefix X /sys/fs/cgroup/cpuset
echo "/sbin/cpuset_release_agent" > /sys/fs/cgroup/cpuset/release_agent
```

2.2 Adding/removing cpus

This is the syntax to use when writing in the cpus or mems files in cpuset directories:

```
# /bin/echo 1-4 > cpuset.cpus         -> set cpus list to cpus 1,2,3,4
# /bin/echo 1,2,3,4 > cpuset.cpus     -> set cpus list to cpus 1,2,3,4
```

To add a CPU to a cpuset, write the new list of CPUs including the CPU to be added. To add 6 to the above cpuset:

```
# /bin/echo 1-4,6 > cpuset.cpus       -> set cpus list to cpus 1,2,3,4,6
```

Similarly to remove a CPU from a cpuset, write the new list of CPUs without the CPU to be removed.

To remove all the CPUs:

```
# /bin/echo "" > cpuset.cpus          -> clear cpus list
```

2.3 Setting flags

The syntax is very simple:

```
# /bin/echo 1 > cpuset.cpu_exclusive  -> set flag 'cpuset.cpu_exclusive'
# /bin/echo 0 > cpuset.cpu_exclusive  -> unset flag 'cpuset.cpu_exclusive'
```

2.4 Attaching processes

```
# /bin/echo PID > tasks
```

Note that it is PID, not PIDs. You can only attach ONE task at a time. If you have several tasks to attach, you have to do it one after another:

```
# /bin/echo PID1 > tasks
# /bin/echo PID2 > tasks
      ...
# /bin/echo PIDn > tasks
```

3. Questions

Q:

what's up with this '/bin/echo' ?

A:

bash's builtin 'echo' command does not check calls to write() against errors. If you use it in the cpuset file system, you won't be able to tell whether a command succeeded or failed.

Q:

When I attach processes, only the first of the line gets really attached !

A:

We can only return one error code per call to write(). So you should also put only ONE pid.

4. Contact

Web: http://www.bullopensource.org/cpuset