Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'cpuset.7' command

$ man cpuset.7

CPUSET(7) Linux Programmer's Manual CPUSET(7)

NAME

cpuset - confine processes to processor and memory node subsets

DESCRIPTION

The cpuset filesystem is a pseudo-filesystem interface to the kernel cpuset mechanism, which is used to control the processor placement and memory placement of processes. It is commonly mounted at /dev/cpuset.

On systems with kernels compiled with built in support for cpusets, all processes are attached to a cpuset, and cpusets are always present. If a system supports cpusets, then it will have the entry nodev cpuset in the file /proc/filesystems. By mounting the cpuset filesystem (see the EXAMPLES section below), the administrator can configure the cpusets on a system to control the processor and memory placement of processes on that system. By default, if the cpuset configuration on a system is not modified or if the cpuset filesystem is not even mounted, then the cpuset mechanism, though present, has no effect on the system's behav? ior.

A cpuset defines a list of CPUs and memory nodes.

The CPUs of a system include all the logical processing units on which a process can execute, including, if present, multiple processor cores within a package and Hyper-Threads within a processor core. Memory nodes include all distinct banks of main memory; small and SMP systems typically have just one memory node that contains all the system's main

memory, while NUMA (non-uniform memory access) systems have multiple memory nodes.

Cpusets are represented as directories in a hierarchical pseudo-filesystem, where the top directory in the hierarchy (/dev/cpuset) rep? resents the entire system (all online CPUs and memory nodes) and any cpuset that is the child (descendant) of another parent cpuset contains a subset of that parent's CPUs and memory nodes. The directories and files representing cpusets have normal filesystem permissions.

Every process in the system belongs to exactly one cpuset. A process is confined to run only on the CPUs in the cpuset it belongs to, and to allocate memory only on the memory nodes in that cpuset. When a process fork(2)s, the child process is placed in the same cpuset as its parent. With sufficient privilege, a process may be moved from one cpuset to another and the allowed CPUs and memory nodes of an existing cpuset may be changed.

When the system begins booting, a single cpuset is defined that in? cludes all CPUs and memory nodes on the system, and all processes are in that cpuset. During the boot process, or later during normal system operation, other cpusets may be created, as subdirectories of this top cpuset, under the control of the system administrator, and processes may be placed in these other cpusets.

Cpusets are integrated with the sched_setaffinity(2) scheduling affin? ity mechanism and the mbind(2) and set_mempolicy(2) memory-placement mechanisms in the kernel. Neither of these mechanisms let a process make use of a CPU or memory node that is not allowed by that process's cpuset. If changes to a process's cpuset placement conflict with these other mechanisms, then cpuset placement is enforced even if it means overriding these other mechanisms. The kernel accomplishes this over? riding by silently restricting the CPUs and memory nodes requested by these other mechanisms to those allowed by the invoking process's cpuset. This can result in these other calls returning an error, if for example, such a call ends up requesting an empty set of CPUs or memory nodes, after that request is restricted to the invoking

process's cpuset.

Typically, a cpuset is used to manage the CPU and memory-node confine? ment for a set of cooperating processes such as a batch scheduler job, and these other mechanisms are used to manage the placement of individ? ual processes or memory regions within that set or job.

FILES

Each directory below /dev/cpuset represents a cpuset and contains a fixed set of pseudo-files describing the state of that cpuset.

New cpusets are created using the mkdir(2) system call or the mkdir(1) command. The properties of a cpuset, such as its flags, allowed CPUs and memory nodes, and attached processes, are queried and modified by reading or writing to the appropriate file in that cpuset's directory, as listed below.

The pseudo-files in each cpuset directory are automatically created when the cpuset is created, as a result of the mkdir(2) invocation. It is not possible to directly add or remove these pseudo-files.

A cpuset directory that contains no child cpuset directories, and has no attached processes, can be removed using rmdir(2) or rmdir(1). It is not necessary, or possible, to remove the pseudo-files inside the directory before removing it.

The pseudo-files in each cpuset directory are small text files that may be read and written using traditional shell utilities such as cat(1), and echo(1), or from a program by using file I/O library functions or system calls, such as open(2), read(2), write(2), and close(2).

The pseudo-files in a cpuset directory represent internal kernel state and do not have any persistent image on disk. Each of these per-cpuset files is listed and described below.

tasks List of the process IDs (PIDs) of the processes in that cpuset.

The list is formatted as a series of ASCII decimal numbers, each followed by a newline. A process may be added to a cpuset (au? tomatically removing it from the cpuset that previously con? tained it) by writing its PID to that cpuset's tasks file (with or without a trailing newline).

Warning: only one PID may be written to the tasks file at a time. If a string is written that contains more than one PID, only the first one will be used.

notify_on_release

Flag (0 or 1). If set (1), that cpuset will receive special handling after it is released, that is, after all processes cease using it (i.e., terminate or are moved to a different cpuset) and all child cpuset directories have been removed. See the Notify On Release section, below.

cpuset.cpus

List of the physical numbers of the CPUs on which processes in that cpuset are allowed to execute. See List Format below for a description of the format of cpus.

The CPUs allowed to a cpuset may be changed by writing a new list to its cpus file.

cpuset.cpu_exclusive

Flag (0 or 1). If set (1), the cpuset has exclusive use of its CPUs (no sibling or cousin cpuset may overlap CPUs). By de? fault, this is off (0). Newly created cpusets also initially default this to off (0).

Two cpusets are sibling cpusets if they share the same parent cpuset in the /dev/cpuset hierarchy. Two cpusets are cousin cpusets if neither is the ancestor of the other. Regardless of the cpu_exclusive setting, if one cpuset is the ancestor of an? other, and if both of these cpusets have nonempty cpus, then their cpus must overlap, because the cpus of any cpuset are al? ways a subset of the cpus of its parent cpuset.

cpuset.mems

List of memory nodes on which processes in this cpuset are al? lowed to allocate memory. See List Format below for a descrip? tion of the format of mems.

cpuset.mem_exclusive

Flag (0 or 1). If set (1), the cpuset has exclusive use of its

memory nodes (no sibling or cousin may overlap). Also if set (1), the cpuset is a Hardwall cpuset (see below). By default, this is off (0). Newly created cpusets also initially default this to off (0).

Regardless of the mem_exclusive setting, if one cpuset is the ancestor of another, then their memory nodes must overlap, be? cause the memory nodes of any cpuset are always a subset of the memory nodes of that cpuset's parent cpuset.

cpuset.mem_hardwall (since Linux 2.6.26)

Flag (0 or 1). If set (1), the cpuset is a Hardwall cpuset (see below). Unlike mem_exclusive, there is no constraint on whether cpusets marked mem_hardwall may have overlapping memory nodes with sibling or cousin cpusets. By default, this is off (0). Newly created cpusets also initially default this to off (0).

cpuset.memory_migrate (since Linux 2.6.16)

Flag (0 or 1). If set (1), then memory migration is enabled. By default, this is off (0). See the Memory Migration section, below.

cpuset.memory_pressure (since Linux 2.6.16)

A measure of how much memory pressure the processes in this cpuset are causing. See the Memory Pressure section, below. Unless memory_pressure_enabled is enabled, always has value zero (0). This file is read-only. See the WARNINGS section, below.

cpuset.memory_pressure_enabled (since Linux 2.6.16)

Flag (0 or 1). This file is present only in the root cpuset, normally /dev/cpuset. If set (1), the memory_pressure calcula? tions are enabled for all cpusets in the system. By default, this is off (0). See the Memory Pressure section, below.

cpuset.memory_spread_page (since Linux 2.6.17)

Flag (0 or 1). If set (1), pages in the kernel page cache (filesystem buffers) are uniformly spread across the cpuset. By default, this is off (0) in the top cpuset, and inherited from the parent cpuset in newly created cpusets. See the Memory

Spread section, below.

cpuset.memory_spread_slab (since Linux 2.6.17)

Flag (0 or 1). If set (1), the kernel slab caches for file I/O (directory and inode structures) are uniformly spread across the cpuset. By default, is off (0) in the top cpuset, and inherited from the parent cpuset in newly created cpusets. See the Memory Spread section, below.

cpuset.sched_load_balance (since Linux 2.6.24)

Flag (0 or 1). If set (1, the default) the kernel will automatically load balance processes in that cpuset over the allowed CPUs in that cpuset. If cleared (0) the kernel will avoid load balancing processes in this cpuset, unless some other cpuset with overlapping CPUs has its sched_load_balance flag set. See Scheduler Load Balancing, below, for further details.

cpuset.sched_relax_domain_level (since Linux 2.6.26)

Integer, between -1 and a small positive value. The sched_relax_domain_level controls the width of the range of CPUs over which the kernel scheduler performs immediate rebalancing of runnable tasks across CPUs. If sched_load_balance is disabled, then the setting of sched_relax_domain_level does not matter, as no such load balancing is done. If sched_load_balance is enabled, then the higher the value of the sched_relax_domain_level, the wider the range of CPUs over which immediate load balancing is attempted. See Scheduler Relax Domain Level, below, for further details.

In addition to the above pseudo-files in each directory below /dev/cpuset, each process has a pseudo-file, /proc/<pid>/cpuset, that displays the path of the process's cpuset directory relative to the root of the cpuset filesystem.

Also the /proc/<pid>/status file for each process has four added lines, displaying the process'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 Mask Format and List Format (see below) as shown in the

following example:

Cpus_allowed: ffffffff,ffffffff,ffffffff,ffffffff

Cpus_allowed_list: 0-127

Mems_allowed: ffffffff,ffffffff

Mems_allowed_list: 0-63

The "allowed" fields were added in Linux 2.6.24; the "allowed_list" fields were added in Linux 2.6.26.

EXTENDED CAPABILITIES

In addition to controlling which cpus and mems a process is allowed to use, cpusets provide the following extended capabilities.

Exclusive cpusets

If a cpuset is marked cpu_exclusive 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 mem_exclusive restricts kernel allocations for buffer cache pages and other internal kernel data pages commonly shared by the kernel across multiple users. All cpusets, whether mem_exclusive or not, restrict allocations of memory for user space. This enables con? figuring a system so that several independent jobs can share common kernel data, 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 indi? vidual job. Only a small amount of kernel memory, such as requests from interrupt handlers, is allowed to be placed on memory nodes out? side even a mem_exclusive cpuset.

Hardwall

A cpuset that has mem_exclusive or mem_hardwall set is a hardwall cpuset. A hardwall cpuset restricts kernel allocations for page, buf? fer, and other data commonly shared by the kernel across multiple users. All cpusets, whether hardwall 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 filesystem pages, while isolating

each job's user allocation in its own cpuset. To do this, construct a large hardwall cpuset to hold all the jobs, and construct child cpusets for each individual job which are not hardwall cpusets.

Only a small amount of kernel memory, such as requests from interrupt handlers, is allowed to be taken outside even a hardwall cpuset.

Notify on release

If the notify_on_release flag is enabled (1) in a cpuset, then whenever the last process in the cpuset leaves (exits or attaches to some other cpuset) and the last child cpuset of that cpuset is removed, the kernel will run the command /sbin/cpuset_release_agent, supplying the pathname (relative to the mount point of the cpuset filesystem) of the abandoned cpuset. This enables automatic removal of abandoned cpusets.

The default value of notify_on_release in the root cpuset at system boot is disabled (0). The default value of other cpusets at creation is the current value of their parent's notify_on_release setting.

The command /sbin/cpuset_release_agent is invoked, with the name (/dev/cpuset relative path) of the to-be-released cpuset in argv[1].

The usual contents of the command /sbin/cpuset_release_agent is simply the shell script:

```
#!/bin/sh
rmdir /dev/cpuset/$1
```

As with other flag values below, this flag can be changed by writing an ASCII number 0 or 1 (with optional trailing newline) into the file, to clear or set the flag, respectively.

Memory pressure

The memory_pressure of a cpuset provides a simple per-cpuset running average of the rate that the processes 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 that are monitoring jobs running in dedi? cated 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 reprioritize jobs that are trying to use more memory than allowed on the nodes assigned 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 action to take if it detects signs of memory pressure.

Unless memory pressure calculation is enabled by setting the pseudo-file /dev/cpuset/cpuset.memory_pressure_enabled, it is not computed for any cpuset, and reads from any memory_pressure always return zero, as represented by the ASCII string "0\n". See the WARNINGS section, be? low.

A per-cpuset, running average is employed for the following reasons:

* Because this meter is per-cpuset rather than per-process or per vir? tual memory region, 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 rather than 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-process, the batch scheduler can obtain the key information?memory pressure in a cpuset?with a single read, rather than having to query and accumu? late results over all the (dynamically changing) set of processes in the cpuset.

The memory_pressure of a cpuset is calculated using a per-cpuset simple digital filter that is kept within the kernel. For each cpuset, this filter tracks the recent rate at which processes attached to that cpuset enter the kernel direct reclaim code.

The kernel direct reclaim code is entered whenever a process has to

satisfy a memory page request by first finding some other page to re?
purpose, due to lack of any readily available already free pages.
Dirty filesystem pages are repurposed by first writing them to disk.
Unmodified filesystem buffer pages are repurposed by simply dropping
them, though if that page is needed again, it will have to be reread
from disk.
The cpuset.memory_pressure file provides an integer number representing
the recent (half-life of 10 seconds) rate of entries to the direct re?
claim code caused by any process in the cpuset, in units of reclaims
attempted per second, times 1000.

Memory spread

There are two Boolean flag files per cpuset that control where the ker?
nel allocates pages for the filesystem 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 filesystem buffers (page cache) evenly
over all the nodes that the faulting process is allowed to use, instead
of preferring to put those pages on the node where the process is run?
ning.
If the per-cpuset Boolean flag file cpuset.memory_spread_slab is set,
then the kernel will spread some filesystem-related slab caches, such
as those for inodes and directory entries, evenly over all the nodes
that the faulting process is allowed to use, instead of preferring to
put those pages on the node where the process is running.
The setting of these flags does not affect the data segment (see
brk(2)) or stack segment pages of a process.
By default, both kinds of memory spreading are off and the kernel pre?
fers to allocate memory pages on the node local to where the requesting
process is running. If that node is not allowed by the process's NUMA
memory policy or cpuset configuration or if there are insufficient free
memory pages on that node, then the kernel looks for the nearest node
that is allowed and has sufficient free memory.

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 process's NUMA memory policy and be spread instead. However, the effect of these changes in memory placement caused by cpuset-specified memory spreading is hidden from the mbind(2) or set_mempolicy(2) calls. These two NUMA memory policy calls always appear to behave as if no cpuset-specified memory spreading is in ef? fect, even if it is. If cpuset memory spreading is subsequently turned off, the NUMA memory policy most recently specified by these calls is automatically reapplied.

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

Cpuset-specified memory spreading behaves similarly to what is known (in other contexts) as round-robin or interleave memory placement.

Cpuset-specified memory spreading can provide substantial performance improvements for jobs that:

a) need to place thread-local data on memory nodes close to the CPUs which are running the threads that most frequently access that data; but also

b) need to access large filesystem data sets that must to be spread across the several nodes in the job's cpuset in order to fit.

Without this policy, the memory allocation across the nodes in the job's cpuset can become very uneven, especially for jobs that might have just a single thread initializing or reading in the data set.

Memory migration

Normally, under the default setting (disabled) of cpuset.memory_mi? grate, 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 cpuset's memory-placement policy mems subsequently changes.

When memory migration is enabled in a cpuset, if the mems setting of the cpuset is changed, then any memory page in use by any process in the cpuset that is on a memory node that is no longer allowed will be migrated to a memory node that is allowed.

Furthermore, if a process is moved into a cpuset with memory_migrate enabled, any memory pages it uses that were on memory nodes allowed in its previous cpuset, but which are not allowed in its new cpuset, will be migrated to a memory node allowed in the new cpuset.

The relative placement of a migrated page within the cpuset is pre? served 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, if pos? sible.

Scheduler load balancing

The kernel scheduler automatically load balances processes. If one CPU is underutilized, the kernel will look for processes on other more overloaded CPUs and move those processes to the underutilized CPU, within the constraints of such placement mechanisms as cpusets and sched_setaffinity(2).

The algorithmic cost of load balancing and its impact on key shared kernel data structures such as the process list increases more than linearly with the number of CPUs being balanced. For example, it costs more to load balance across one large set of CPUs than it does to bal? ance across two smaller sets of CPUs, each of half the size of the larger set. (The precise relationship between the number of CPUs being balanced and the cost of load balancing depends on implementation de? tails of the kernel process scheduler, which is subject to change over time, as improved kernel scheduler algorithms are implemented.)

The per-cpuset flag sched_load_balance provides a mechanism to suppress this automatic scheduler load balancing in cases where it is not needed and suppressing it would have worthwhile performance benefits.

By default, load balancing is done across all CPUs, except those marked isolated using the kernel boot time "isolcpus=" argument. (See Sched?

uler Relax Domain Level, below, to change this default.)

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

* 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.
* Systems supporting real-time on some CPUs need to minimize system overhead on those CPUs, including avoiding process load balancing if that is not needed.

When the per-cpuset flag sched_load_balance is enabled (the default setting), it requests load balancing across all the CPUs in that cpuset's allowed CPUs, ensuring that load balancing can move a process (not otherwise pinned, as by sched_setaffinity(2)) from any CPU in that cpuset to any other.

When the per-cpuset flag sched_load_balance is disabled, then the scheduler will avoid load balancing across the CPUs in that cpuset, ex? cept 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 sched_load_balance en? abled, then the scheduler will load balance across all CPUs, and the setting of the sched_load_balance flag in other cpusets has no effect, as we're already fully load balancing.

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

When doing this, you don't usually want to leave any unpinned processes in the top cpuset that might use nontrivial amounts of CPU, as such processes may be artificially constrained to some subset of CPUs, de? pending on the particulars of this flag setting in descendant cpusets. Even if such a process could use spare CPU cycles in some other CPUs, the kernel scheduler might not consider the possibility of load balanc? ing that process to the underused CPU.

Of course, processes pinned to a particular CPU can be left in a cpuset

that disables sched_load_balance as those processes aren't going any? where else anyway.

Scheduler relax domain level

The kernel scheduler performs immediate load balancing whenever a CPU becomes free or another task becomes runnable. This load balancing works to ensure that as many CPUs as possible are usefully employed running tasks. The kernel also performs periodic load balancing off the software clock described in time(7). The setting of sched_re? lax_domain_level applies only to immediate load balancing. Regardless of the sched_relax_domain_level setting, periodic load balancing is at? tempted over all CPUs (unless disabled by turning off sched_load_bal? ance.) In any case, of course, tasks will be scheduled to run only on CPUs allowed by their cpuset, as modified by sched_setaffinity(2) sys? tem calls.

On small systems, such as those with just a few CPUs, immediate load balancing is useful to improve system interactivity and to minimize wasteful idle CPU cycles. But on large systems, attempting immediate load balancing across a large number of CPUs can be more costly than it is worth, depending on the particular performance characteristics of the job mix and the hardware.

The exact meaning of the small integer values of sched_relax_do? main_level will depend on internal implementation details of the kernel scheduler code and on the non-uniform architecture of the hardware. Both of these will evolve over time and vary by system architecture and kernel version.

As of this writing, when this capability was introduced in Linux 2.6.26, on certain popular architectures, the positive values of sched_relax_domain_level have the following meanings.

(1) Perform immediate load balancing across Hyper-Thread siblings on the same core.

(2) Perform immediate load balancing across other cores in the same package.

(3) Perform immediate load balancing across other CPUs on the same node

or blade.

(4) Perform immediate load balancing across over several (implementa? tion detail) nodes [On NUMA systems].

(5) Perform immediate load balancing across over all CPUs in system [On NUMA systems].

The sched_relax_domain_level value of zero (0) always means don't per? form immediate load balancing, hence that load balancing is done only periodically, not immediately when a CPU becomes available or another task becomes runnable.

The sched_relax_domain_level value of minus one (-1) always means use the system default value. The system default value can vary by archi? tecture and kernel version. This system default value can be changed by kernel boot-time "relax_domain_level=" argument.

In the case of multiple overlapping cpusets which have conflicting sched_relax_domain_level values, then the highest such value applies to all CPUs in any of the overlapping cpusets. In such cases, the value minus one (-1) is the lowest value, overridden by any other value, and the value zero (0) is the next lowest value.

FORMATS

The following formats are used to represent sets of CPUs and memory nodes.

Mask format

The Mask Format is used to represent CPU and memory-node bit masks in the /proc/<pid>/status file.

This format displays each 32-bit word in hexadecimal (using ASCII char? acters "0" - "9" and "a" - "f"); words are filled with leading zeros, if required. For masks longer than one word, a comma separator is used between words. Words are displayed in big-endian order, which has the most significant bit first. The hex digits within a word are also in big-endian order.

The number of 32-bit words displayed is the minimum number needed to display all bits of the bit mask, based on the size of the bit mask.

Examples of the Mask Format:

00000001 # just bit 0 set

40000000,00000000,00000000 # just bit 94 set

00000001,00000000,00000000 # just bit 64 set

000000ff,00000000 # bits 32-39 set

00000000,000e3862 # 1,5,6,11-13,17-19 set

A mask with bits 0, 1, 2, 4, 8, 16, 32, and 64 set displays as:

00000001,00000001,00010117

The first "1" is for bit 64, the second for bit 32, the third for bit 16, the fourth for bit 8, the fifth for bit 4, and the "7" is for bits 2, 1, and 0.

List format

The List Format for cpus and mems is a comma-separated list of CPU or memory-node numbers and ranges of numbers, in ASCII decimal.

Examples of the List Format:

0-4,9 # bits 0, 1, 2, 3, 4, and 9 set

0-2,7,12-14 # bits 0, 1, 2, 7, 12, 13, and 14 set

RULES

The following rules apply to each cpuset:

* Its CPUs and memory nodes must be a (possibly equal) subset of its parent's.
* It can be marked cpu_exclusive only if its parent is.
* It can be marked mem_exclusive only if its parent is.
* If it is cpu_exclusive, its CPUs may not overlap any sibling.
* If it is memory_exclusive, its memory nodes may not overlap any sib? ling.

PERMISSIONS

The permissions of a cpuset are determined by the permissions of the directories and pseudo-files in the cpuset filesystem, normally mounted at /dev/cpuset.

For instance, a process can put itself in some other cpuset (than its current one) if it can write the tasks file for that cpuset. This re? quires execute permission on the encompassing directories and write permission on the tasks file.

An additional constraint is applied to requests to place some other process in a cpuset. One process may not attach another to a cpuset unless it would have permission to send that process a signal (see kill(2)).

A process may create a child cpuset if it can access and write the par? ent cpuset directory. It can modify the CPUs or memory nodes in a cpuset if it can access that cpuset's directory (execute permissions on the each of the parent directories) and write the corresponding cpus or mems file.

There is one minor difference between the manner in which these permis? sions are evaluated and the manner in which normal filesystem operation permissions are evaluated. The kernel interprets relative pathnames starting at a process's current working directory. Even if one is op? erating on a cpuset file, relative pathnames are interpreted relative to the process's current working directory, not relative to the process's current cpuset. The only ways that cpuset paths relative to a process's current cpuset can be used are if either the process's cur? rent working directory is its cpuset (it first did a cd or chdir(2) to its cpuset directory beneath /dev/cpuset, which is a bit unusual) or if some user code converts the relative cpuset path to a full filesystem path.

In theory, this means that user code should specify cpusets using abso? lute pathnames, which requires knowing the mount point of the cpuset filesystem (usually, but not necessarily, /dev/cpuset). In practice, all user level code that this author is aware of simply assumes that if the cpuset filesystem is mounted, then it is mounted at /dev/cpuset. Furthermore, it is common practice for carefully written user code to verify the presence of the pseudo-file /dev/cpuset/tasks in order to verify that the cpuset pseudo-filesystem is currently mounted.

WARNINGS

Enabling memory_pressure

By default, the per-cpuset file cpuset.memory_pressure always contains zero (0). Unless this feature is enabled by writing "1" to the pseudo-

file /dev/cpuset/cpuset.memory_pressure_enabled, the kernel does not compute per-cpuset memory_pressure.

Using the echo command

When using the echo command at the shell prompt to change the values of cpuset files, beware that the built-in echo command in some shells does not display an error message if the write(2) system call fails. For example, if the command:

```
echo 19 > cpuset.mems
```

failed because memory node 19 was not allowed (perhaps the current system does not have a memory node 19), then the echo command might not display any error. It is better to use the /bin/echo external command to change cpuset file settings, as this command will display write(2) errors, as in the example:

```
/bin/echo 19 > cpuset.mems
/bin/echo: write error: Invalid argument
```

EXCEPTIONS

Memory placement

Not all allocations of system memory are constrained by cpusets, for the following reasons.

If hot-plug functionality is used to remove all the CPUs that are currently assigned to a cpuset, then the kernel will automatically update the cpus_allowed of all processes attached to CPUs in that cpuset to allow all CPUs. When memory hot-plug 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, rather than starving a process that has had all its allowed CPUs or memory nodes taken offline. User code should reconfigure cpusets to refer only to online CPUs and memory nodes when using hot-plug to add or remove such resources.

A few kernel-critical, internal memory-allocation requests, marked GFP_ATOMIC, must be satisfied immediately. The kernel may drop some request or malfunction if one of these allocations fail. If such a request cannot be satisfied within the current process'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.

Allocations of memory requested by kernel drivers while processing an interrupt lack any relevant process context, and are not confined by cpusets.

Renaming cpusets

You can use the rename(2) system call to rename cpusets. Only simple renaming is supported; that is, changing the name of a cpuset directory is permitted, but moving a directory into a different directory is not permitted.

ERRORS

The Linux kernel implementation of cpusets sets errno to specify the reason for a failed system call affecting cpusets.

The possible errno settings and their meaning when set on a failed cpuset call are as listed below.

E2BIG Attempted a write(2) on a special cpuset file with a length larger than some kernel-determined upper limit on the length of such writes.

EACCES Attempted to write(2) the process ID (PID) of a process to a cpuset tasks file when one lacks permission to move that process.

EACCES Attempted to add, using write(2), a CPU or memory node to a cpuset, when that CPU or memory node was not already in its par? ent.

EACCES Attempted to set, using write(2), cpuset.cpu_exclusive or cpuset.mem_exclusive on a cpuset whose parent lacks the same setting.

EACCES Attempted to write(2) a cpuset.memory_pressure file.

EACCES Attempted to create a file in a cpuset directory.

EBUSY Attempted to remove, using rmdir(2), a cpuset with attached pro? cesses.

EBUSY Attempted to remove, using rmdir(2), a cpuset with child cpusets.

EBUSY Attempted to remove a CPU or memory node from a cpuset that is also in a child of that cpuset.

EEXIST Attempted to create, using mkdir(2), a cpuset that already ex? ists.

EEXIST Attempted to rename(2) a cpuset to a name that already exists.

EFAULT Attempted to read(2) or write(2) a cpuset file using a buffer that is outside the writing processes accessible address space.

EINVAL Attempted to change a cpuset, using write(2), in a way that would violate a cpu_exclusive or mem_exclusive attribute of that cpuset or any of its siblings.

EINVAL Attempted to write(2) an empty cpuset.cpus or cpuset.mems list to a cpuset which has attached processes or child cpusets.

EINVAL Attempted to write(2) a cpuset.cpus or cpuset.mems list which included a range with the second number smaller than the first number.

EINVAL Attempted to write(2) a cpuset.cpus or cpuset.mems list which included an invalid character in the string.

EINVAL Attempted to write(2) a list to a cpuset.cpus file that did not include any online CPUs.

EINVAL Attempted to write(2) a list to a cpuset.mems file that did not include any online memory nodes.

EINVAL Attempted to write(2) a list to a cpuset.mems file that included a node that held no memory.

EIO Attempted to write(2) a string to a cpuset tasks file that does not begin with an ASCII decimal integer.

EIO Attempted to rename(2) a cpuset into a different directory.

ENAMETOOLONG

Attempted to read(2) a /proc/<pid>/cpuset file for a cpuset path that is longer than the kernel page size.

ENAMETOOLONG

Attempted to create, using mkdir(2), a cpuset whose base direc? tory name is longer than 255 characters.

ENAMETOOLONG

Attempted to create, using mkdir(2), a cpuset whose full path? name, including the mount point (typically "/dev/cpuset/") pre? fix, is longer than 4095 characters.

ENODEV The cpuset was removed by another process at the same time as a write(2) was attempted on one of the pseudo-files in the cpuset directory.

ENOENT Attempted to create, using mkdir(2), a cpuset in a parent cpuset that doesn't exist.

ENOENT Attempted to access(2) or open(2) a nonexistent file in a cpuset directory.

ENOMEM Insufficient memory is available within the kernel; can occur on a variety of system calls affecting cpusets, but only if the system is extremely short of memory.

ENOSPC Attempted to write(2) the process ID (PID) of a process to a cpuset tasks file when the cpuset had an empty cpuset.cpus or empty cpuset.mems setting.

ENOSPC Attempted to write(2) an empty cpuset.cpus or cpuset.mems set? ting to a cpuset that has tasks attached.

ENOTDIR

Attempted to rename(2) a nonexistent cpuset.

EPERM Attempted to remove a file from a cpuset directory.

ERANGE Specified a cpuset.cpus or cpuset.mems list to the kernel which included a number too large for the kernel to set in its bit masks.

ESRCH Attempted to write(2) the process ID (PID) of a nonexistent process to a cpuset tasks file.

VERSIONS

Cpusets appeared in version 2.6.12 of the Linux kernel.

NOTES

Despite its name, the pid parameter is actually a thread ID, and each thread in a threaded group can be attached to a different cpuset. The value returned from a call to gettid(2) can be passed in the argument pid.

BUGS

cpuset.memory_pressure cpuset files can be opened for writing, cre?
ation, or truncation, but then the write(2) fails with errno set to
EACCES, and the creation and truncation options on open(2) have no ef?
fect.

EXAMPLES

The following examples demonstrate querying and setting cpuset options
using shell commands.

Creating and attaching to a cpuset.

To create a new cpuset and attach the current command shell to it, the
steps are:

1) mkdir /dev/cpuset (if not already done)

2) mount -t cpuset none /dev/cpuset (if not already done)

3) Create the new cpuset using mkdir(1).

4) Assign CPUs and memory nodes to the new cpuset.

5) Attach the shell to the new cpuset.

For example, the following sequence of commands will set up a cpuset
named "Charlie", containing just CPUs 2 and 3, and memory node 1, and
then attach the current shell to that cpuset.

```
$ mkdir /dev/cpuset
$ mount -t cpuset cpuset /dev/cpuset
$ cd /dev/cpuset
$ mkdir Charlie
$ cd Charlie
$ /bin/echo 2-3 > cpuset.cpus
$ /bin/echo 1 > cpuset.mems
$ /bin/echo $$ > tasks
# The current shell is now running in cpuset Charlie
# The next line should display '/Charlie'
$ cat /proc/self/cpuset
```

Migrating a job to different memory nodes.

To migrate a job (the set of processes attached to a cpuset) to differ?
ent CPUs and memory nodes in the system, including moving the memory

pages currently allocated to that job, perform the following steps.

1) Let's say we want to move the job in cpuset alpha (CPUs 4?7 and memory nodes 2?3) to a new cpuset beta (CPUs 16?19 and memory nodes 8?9).

2) First create the new cpuset beta.

3) Then allow CPUs 16?19 and memory nodes 8?9 in beta.

4) Then enable memory_migration in beta.

5) Then move each process from alpha to beta.

The following sequence of commands accomplishes this.

```
$ cd /dev/cpuset
$ mkdir beta
$ cd beta
$ /bin/echo 16-19 > cpuset.cpus
$ /bin/echo 8-9 > cpuset.mems
$ /bin/echo 1 > cpuset.memory_migrate
$ while read i; do /bin/echo $i; done < ../alpha/tasks > tasks
```

The above should move any processes in alpha to beta, and any memory held by these processes on memory nodes 2?3 to memory nodes 8?9, re?spectively.

Notice that the last step of the above sequence did not do:

```
$ cp ../alpha/tasks tasks
```

The while loop, rather than the seemingly easier use of the cp(1) com?mand, was necessary because only one process PID at a time may be writ?ten to the tasks file.

The same effect (writing one PID at a time) as the while loop can be accomplished more efficiently, in fewer keystrokes and in syntax that works on any shell, but alas more obscurely, by using the -u (un?buffered) option of sed(1):

```
$ sed -un p < ../alpha/tasks > tasks
```

SEE ALSO

taskset(1), get_mempolicy(2), getcpu(2), mbind(2), sched_getaffin?ity(2), sched_setaffinity(2), sched_setscheduler(2), set_mempolicy(2), CPU_SET(3), proc(5), cgroups(7), numa(7), sched(7), migratepages(8),

numactl(8)

Documentation/admin-guide/cgroup-v1/cpusets.rst in the Linux kernel source tree (or Documentation/cgroup-v1/cpusets.txt before Linux 4.18, and Documentation/cpusets.txt before Linux 2.6.29)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.